THESIS

DATA MINING OF EXTREMELY LARGE AD HOC DATA SETS TO PRODUCE INVERTED INDICES

by

Aaron D. Coudray

June 2016

Thesis Advisor:	Frank Kragh
Co-Advisor:	Jim Scrofani

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** June 2016	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE DATA MINING OF EXTREMELY LARGE AD HOC DATA SETS TO PRODUCE INVERTED INDICES			**5. FUNDING NUMBERS**
6. AUTHOR(S) Aaron D. Coudray			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

The purpose of this study is to leverage existing Internet-sized ad hoc data sets by creating an inverted index that will enable a robust search capability. In particular, this study is focused on the Common Crawl web corpus. This involves exploring the tools and techniques necessary to effectively traverse this data set, as well as producing the tools to create an inverted index relationship between the terms and websites found within web archive files. The primary tools utilized in this process are Apache Hadoop, Apache MapReduce, Amazon Web Services, and Java. Additionally, methods to enhance this relationship with other information of interest are investigated in this thesis. Specifically, an index was developed that contains the added component of term relative location. This inverted index relationship is an essential component of—and the first step in—creating a robust search capability for a very large ad hoc data set.

14. SUBJECT TERMS big data, Common Crawl, Hadoop, inverted index, inverted indices, Java, MapReduce			**15. NUMBER OF PAGES** 167
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

DATA MINING OF EXTREMELY LARGE AD HOC DATA SETS TO PRODUCE INVERTED INDICES

Aaron D. Coudray
Lieutenant Commander, United States Navy
B.S., Tulane University, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 2016

Approved by: Frank Kragh
Thesis Advisor

Jim Scrofani
Co-Advisor

Clark Robertson
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The purpose of this study is to leverage existing Internet-sized ad hoc data sets by creating an inverted index that will enable a robust search capability. In particular, this study is focused on the Common Crawl web corpus. This involves exploring the tools and techniques necessary to effectively traverse this data set, as well as producing the tools to create an inverted index relationship between the terms and websites found within web archive files. The primary tools utilized in this process are Apache Hadoop, Apache MapReduce, Amazon Web Services, and Java. Additionally, methods to enhance this relationship with other information of interest are investigated in this thesis. Specifically, an index was developed that contains the added component of term relative location. This inverted index relationship is an essential component of—and the first step in—creating a robust search capability for a very large ad hoc data set.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

API	application program interface
ARC	archive
AWS	Amazon web services
CLI	command line interface
CSS	cascading style sheets
EC2	elastic compute cloud
EMR	elastic MapReduce
GB	gigabyte
GFS	Google file system
GZIP	GNU zip
HDFS	Hadoop distributed file system
HTML	hypertext markup language
HTTP	hypertext transfer protocol
IDE	integrated development environment
ISO	international organization for standardization
JAR	Java archive
JSON	JavaScript object notation
JVM	Java virtual machine
JWAT	Java web archive toolkit
MB	megabyte
OS	operating system
POM	project object model
RDFa	resource description framework in attributes
RFC	request for comments
S3	simple storage system
SQL	structured query language
SSH	secure shell
TB	terabyte
URL	uniform resource locator
VM	virtual machine

WARC	web archive
WAT	web archive transformation file
WET	WinSQL export template file
WWW	world wide web
XML	extensible markup language

ACKNOWLEDGMENTS

I would like to thank my beautiful bride and better half for her always loving me and supporting me. God has blessed my life with an amazing friend, wife, and mother in her—without whom I'd be lost. I would also like to thank my advisor, Professor Frank Kragh, for his support and guidance through this process.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

We live in an age dominated by information technology. Computers and other devices have gradually become more pervasive over the years, but have exploded in availability and usage within the past decade. For example, the birth of the smartphone occurred in 2007 when the first iPhone was released; now, in 2016, there are people who feel unable to function without one. Among the key elements enabling this rapid rise in the use of information devices is the improvement of storage technology. Individuals and organizations can now collect and store large quantities of data at an affordable cost, which has resulted in an unprecedented volume of data being maintained. There are some estimates that over 90 percent of the world's data has been created in the past two years [1].

Not only are people storing greater amounts of data, but they also have greater accessibility to the data being stored. As technological capabilities have grown and become more affordable, the connectedness between people has increased. One of the easiest ways to share information is through the Internet. This capability has increased so much that there are now institutions and other organizations looking for ways to harness the continuously expanding availability of data for learning and other research projects. One such organization is called Common Crawl, a non-profit organization that for the past seven years has been providing access to web crawl data to the public [2]. As the years have passed, Common Crawl has continued to increase the frequency at which updated data is provided, currently every month. This data is freely available on Amazon Web Services (AWS) Simple Storage System (S3) [2].

B. OBJECTIVE

The purpose of the work described in this thesis is to investigate the Common Crawl data, specifically to determine the tools necessary to effectively traverse this data, as well as develop the tools and techniques required to explore and use this data in support of government operations. The two fundamental areas that are the foundation for this project consist of the data format and the architecture in which the data resides, both of which directly influence how data is accessed. More specifically, this involved an understanding of a Web ARChive (WARC) file (the format) and the Apache Hadoop framework (the architecture).

The ultimate goal of this research project is to create a very robust search capability for the crawl data. In its simplest form, this is the ability to conduct a word or phrase search and obtain related results; thus, the primary accomplishment of this thesis will be the first step in the process toward developing this search capability. Specifically, this first step will be to develop the ability to create an inverted index of the crawl data, taking the dataset of interest and creating the relationship of words and the websites on which these words can be found.

The ability to perform word and phrase searches on archived web data is a very useful capability. This type of search produces results not necessarily obtained using a standard search engine such as Google or Yahoo! Contained within the WARC file is the HyperText Markup Language (HTML) of each website crawled, as well as metadata from the crawl itself [3]. This provides access to not only the plain text of the website but other potentially useful information (e.g., geo-location of site, author of site, date of creation/modification). This additional information, or meta-data, of interest is not a focus of a typical search engine; in addition, these types of search engines are designed to be useful to the average user and have built-in ranking algorithms to improve search results. These types of algorithms (e.g., PageRank) help to promote webpages that are deemed more important and more likely to have useful results. From the perspective of this research, the most interesting results are not necessarily synonymous with the most important results determined by a typical search engine. Some of the more interesting results can potentially be contained in forum posts or other user commentary on a multitude of websites.

C. ORGANIZATION

This thesis is organized into six chapters. A high-level discussion of the problem and objective of this research is contained in Chapter I. A discussion of the overarching concepts related to this project, as well as the tools and services necessary for tackling this problem, is provided in Chapter II. A detailed discussion of the search method and execution for the crawl data is discussed in Chapter III. The construction of inverted index portion of this search tool is contained in Chapter IV. Analyses of the associated results are presented in Chapter V. Finally, a summary of the work performed in conjunction with this thesis and recommendations for future work are presented in Chapter VI.

THIS PAGE INTENTIONALLY LEFT BLANK

II. OVERVIEW

A. BIG DATA

The Merriam-Webster dictionary defines big data as “an accumulation of data that is too large and complex for processing by traditional database management tools” [4]. This is a fairly generic definition and may be interpreted differently depending on the application. In the early 2000s, Doug Laney first proposed the three Vs, which are criteria that have since become the foundation for defining big data [5]. The three Vs are:

1. Volume: The amount of data.
2. Velocity: The rate at which data is produced or processed.
3. Variety: The type(s) of data.

Recently, two more Vs have been included (i.e., Veracity and Value) in efforts to express additional desirable characteristics for big data [6]; however, these two additional characteristics do not fundamentally change the foundational qualities separating big data analysis from classical data analysis. Even with the original three Vs, the determination of whether a dataset is considered big data depends on what tools are available. Ten years ago, a large volume of data may have been on the order of a terabyte (TB) (i.e., one trillion bytes), but today it is common to have multiple TBs of storage for personal use. Large volume in today’s terms is more likely to be on the petabyte (i.e., 1,000 TBs), or even the exabyte (i.e., one million TBs), scale.

For the purposes of this thesis, the fundamental characteristic that makes this problem a big data problem is the size of the dataset. Not only is the Common Crawl dataset large, but it is also located within a distributed system. This data, therefore, is a challenge to analyze in that it is difficult to load into one machine as well as balancing the cost, latency, and inefficiency of moving that much data around. Because of these challenges, the classical methods of data mining and analysis do not work. Thankfully, as the data has grown, the tools used for analysis have evolved and are still evolving to this day.

B. HADOOP DISTRIBUTED FILE SYSTEM (HDFS)

Unsurprisingly, the ability to store large volumes of data has resulted in other challenges. One of these challenges has been to ensure long-term availability of that data. For an individual user, the failure of a single computer is relatively unlikely; however, when data becomes so large that is requires a storage system comprised of hundreds or even thousands of machines, failures within that system are no longer the exception—failures can and will happen. For example, let us assume a single server can stay operational for three years (approximately 1,000 days); then, with a cluster of 1,000 servers, this results in approximately one failure per day. It follows then, that if this same system were expanded to a cluster of one million servers, this equates to 1,000 failures per day. If a system is susceptible to this magnitude failure rate, then data will be lost unless there is a distributed file system that ensures the availability of the data.

In 2003, a paper emerged from Google which discussed the Google File System (GFS), a solution to creating a scalable distributed file system [7]. The GFS was designed to meet Google’s data scalability and reliability objectives based on the data usage characteristics observed within their systems. Around the same time, Doug Cutting and Mike Cafarella were working on an independent project called Nutch. These two also were attempting to overcome data scalability issues, and Google’s paper regarding the GFS became pivotal in helping to work through some of their own architectural issues and ultimately laying the groundwork for what is today known as Hadoop [8].

The Hadoop Distributed File System (HDFS) was initially designed to replicate what the GFS had accomplished [8]. One of the key elements of this architecture is the ability to handle component failure without losing data, ensuring the long-term availability of data. The main feature that ensures data reliability is in how data blocks are created and managed within the HDFS. Stored data is typically broken up into 128 megabyte (MB) blocks and stored three times within the HDFS cluster. These blocks of data are then managed by the *namenode* and the *datanodes*. The *namenode* keeps track of the metadata and structure of the HDFS, while the *datanodes* manage the blocks of data within the HDFS. This typical write process is shown in Figure 1 [8].

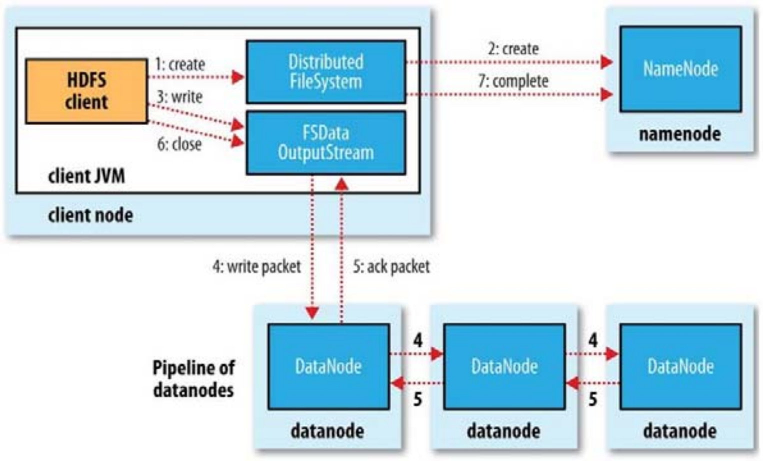


Figure 1. Typical Data Write to HDFS. Source: [8].

The way data blocks are replicated and placed is a tradeoff between data redundancy and system bandwidth. Typically, two of the three blocks are contained within the same rack. If a node failure occurs, this scheme allows for improved read and write bandwidth [8]; however, this scheme alone does not maintain data redundancy in the event of a rack failure, and a third block is placed on a separate rack [8]. This replication scheme is illustrated in Figure 2. Ultimately, HDFS does a good job of ensuring the availability of the data; however, the other challenge of big data is in the ability to analyze the data.

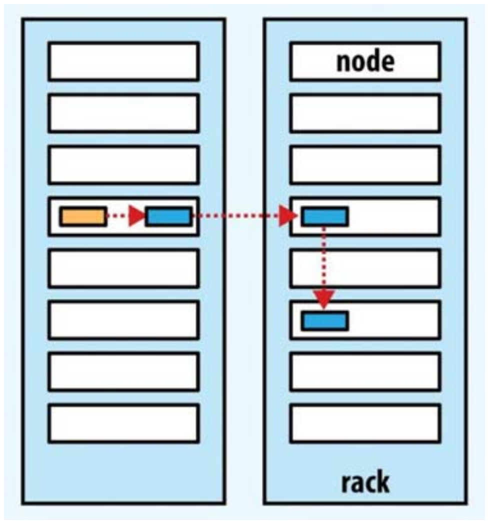


Figure 2. Typical Data Replication and Storage. Source: [8].

C. MAPREDUCE

As stated, another challenge that arises when dealing with big data is the ability to analyze the data. Traditionally, data analysis occurs within a single machine. When dealing with large volumes of data, this is infeasible for multiple reasons. Either the data is too large to fit on a single machine or it is too inefficient to analyze on a single machine due to system bandwidth. For example, let us assume that there is a 200 TB dataset ready for analysis residing on a system with an average disk-read bandwidth of 100 MB per second. This dataset would take approximately two million seconds—or just over 23 days—to read. This same principle can be applied on a larger scale, limited only by the network bandwidth. In either instance, the bandwidth constraint results in prolonged analysis with a single machine and demonstrates the need to have a method to perform distributed computing.

Shortly after Google published its paper on the GFS, it published another paper written to address the distributed processing issue [9]. This paper is about a programming model called MapReduce, designed specifically for use on the GFS for processing large datasets. Just as before, this paper was utilized by the creators of the Nutch project and was the foundation for what is today known as Hadoop MapReduce [8].

The purpose of the MapReduce programming model is to take a large job and process smaller chunks of that job in parallel; this job flow is illustrated in Figure 3. Overall, there are two basic elements to a job, the first stage being the Map portion and the second the Reduce portion; within each phase, data is organized into key-value pairs. The input data is first split up into smaller independent chunks of data. Each of these smaller chunks is processed by a parallel Map function into key-value pairs. Once the Mappers are complete, the intermediate output of the Mappers is organized by the key, at which point the Reducer combines all the results into the final key-value pairs [8].

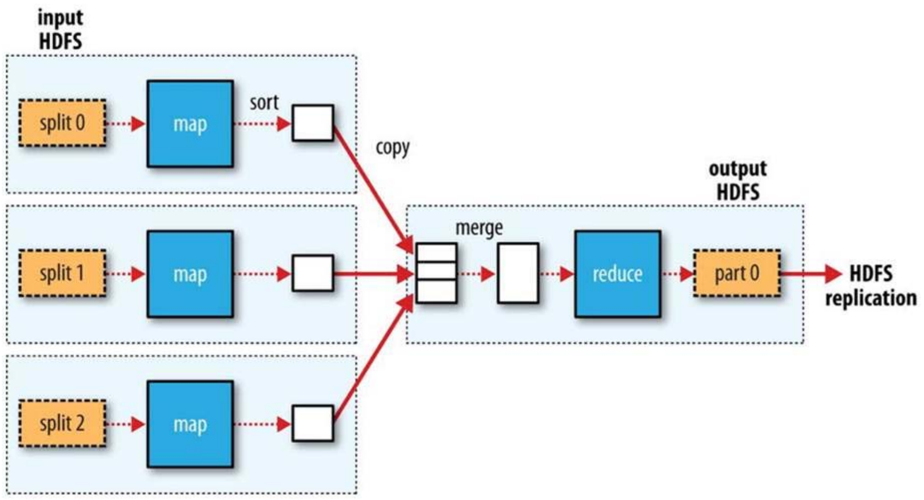


Figure 3. Typical MapReduce Data Flow. Source: [8].

A simple and common example used to demonstrate this capability is that of a word count. The goal of a word count is to take an input set of documents and determine all the words contained within the specified documents as well as the frequency of the words. For example [9]:

```
Mapper (String key, String value):
// key:   the document name
// value: the document contents
     for each word contained in value:
         emit intermediate key-value pair (word, “1”);

Reducer (String key, Iterator values[count1, count2,
count3, …]):
// key:    a word
// values: a list of counts
     int sum = 0;
     for each count in the list of values:
         sum = sum + count;
     emit final key-value pair (word, sum)
```

The Map function takes each individual word and emits with it a value of *one*. The Reduce function then receives a given word and an associated list of *ones*, one for each time the word occurred. The Reducer then sums those occurrences, providing a final count across all documents searched. This simple example demonstrates the parallel computing power of the MapReduce programming model [9].

MapReduce, in conjunction with Hadoop, is the foundation of what is today known as the Apache Hadoop framework. MapReduce makes it significantly easier to write good code for a programmer who may not be familiar with how to handle parallelization and load balancing [8]; however, it still requires programming knowledge, specifically with the Java programming language. While there are many organizations that can benefit from the capabilities of the Hadoop framework, some individual may not be Java programmers or have much programming experience; therefore, other tools have been developed that fall under the umbrella of what is commonly referred to as the Hadoop Ecosystem. Additionally, a utility exists called Hadoop Streaming, which allows for the use of scripting languages such as Python [10]. A small subset of the existing ecosystem is depicted in Figure 4.

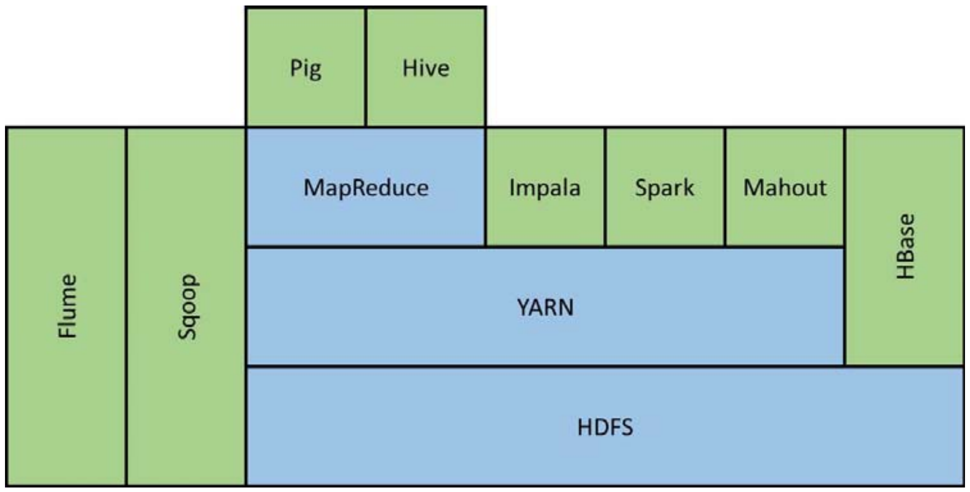


Figure 4. Simplified Diagram of the Hadoop Ecosystem. Adapted from [11].

The tools in this subset range from providing a way to make it easier for people who have experience with Structured Query Language (SQL) or other relational databases (e.g., Hive), rather than the core Hadoop framework, to tools that provide the ability to use a scripting language rather than a programming language like Java to write MapReduce jobs (e.g., Pig). Additionally, there are tools to ingest data from other forms into HDFS (e.g., Flume and Sqoop). While all these tools enhance the robustness of the overall system, ultimately a Hadoop system is still based on the fundamental components of MapReduce and HDFS [8].

D. COMMON CRAWL

The Common Crawl is a non-profit organization that provides regular copies of web crawl data in order to support research and educational endeavors [2]. This vast repository of data is obtained through the use of Nutch-based web crawlers and is made publically available on the Amazon S3 service. Web crawling is a method to index the web and is executed by a multitude of organizations; the most well-known purpose is in support of search engines. This indexing is essentially accomplished by crawling bots that visit a list of websites, search those sites for links to other websites, and any links found are added to the list of sites to visit [2], [12].

To date, the Common Crawl has provided approximately seven years of web crawls, which contains multiple petabytes of data [2]. From 2008 to 2012, three crawls were performed and are available in the Archive (ARC) file format. Starting in 2013, the frequency with which new crawl data was made available increased and is currently at the rate of once per month. Additionally, in 2013, the data switched to the WARC file format, and the organization began providing two additional data files. The additional files are subsets of the WARC file, the first being the WET file, which stands for WinSQL Export Template file, the second being the WAT file, which stands for Web Archive Transformation file [2]. A single web crawl is comprised of many WARC files that are organized into segments. A segment refers to a grouping of Uniform Resource Locators (URL) that are crawled as a set [12]. The exact number and size of individual WARC files, as well as segments, are a result of the overall web crawl configuration.

E. AMAZON WEB SERVICES

Amazon Web Services is an entity within Amazon.com that provides a multitude of services centered on cloud computing (e.g., computing, storage, applications) [13]. With respect to this thesis, its specific products of interest are the S3 and Elastic Compute Cloud (EC2) services. The S3 service provides for long-term secure storage of data, while the EC2 service provides a customizable cloud computing resource. Within these services, the individual files of the Common Crawl data are freely available for download. Because of the size of the entire crawl file set, large-scale analysis cannot be accomplished on an individual machine; therefore, this must be accomplished by downloading files to a distributed system or by using AWS.

F. COMMON CRAWL FILE FORMATS (WARC, WET, WAT)

The WARC file is the base file created by the web crawl. The format for this file is governed by the international standard ISO 28500 (Information and documentation–The WARC File Format) [3]. The primary purpose of the WARC file is for the storage of web crawls. Specifically, it was designed to provide a convention for the concatenation of numerous records as well as providing standard header information. In essence, a WARC record consists of two main elements. The first element is the record header, and the second is the record content. There are numerous potential named fields that may be contained within any given entry, not all of which are required. Additionally, there are ways to prevent a website from responding to web crawlers. Overall, there are multiple types of records contained within each file, not all of which are website responses; as such, not every entry within a WARC file is ultimately of interest [3].

The WET file is generated from the original WARC file. This file contains the extracted plaintext from the WARC record and associated header information in order to identify the originating record. The WAT file contains the metadata about the WARC records stored in the JSON (JavaScript Object Notation) format. The WET and WAT files do not contain any new information outside of the originating WARC file but rather are focused subsets of data generated for ease of analysis [2].

G. WEB SEARCH

Developing a web search algorithm is a quintessential big data problem. It is not surprising that significant contributions in the area of large-scale data storage and processing originated from Google, arguably the provider of the most prominent search engine today. A web search can be broken down into three separate sub-problems: gathering the information, indexing the information, and ranking the information for retrieval [14].

1. Gathering the Data

Crawling bots perform the process of collecting the content contained on the World Wide Web (WWW). This was briefly discussed within the section regarding the Common Crawl organization. The result of this crawling process is what becomes the input for the next phase of the search algorithm, which generates the inverted index. With respect to the Common Crawl, the specific outputs of the web crawl are the WARC, WET, and WAT files [2].

2. Indexing the Data

The inverted index is the element that contains all the information regarding a term and where it is located. A simple illustration of an inverted index is the index contained within a textbook. A textbook index shows the relationship between various topics contained within the book and the respective pages where those topics can be found. If this concept is expanded to a set of books or documents, then an index could represent the relationship between topics or words and all the documents in which they can be found (e.g., a library catalog). This is demonstrated by the following example [14]:

```
Mapper (String key, String value):
// key:   the document name
// value: the document contents
     for each word contained in value:
         emit intermediate key-value pair (word, key);

Reducer (String key, Iterator values[doc1, doc2, doc3, …]):
// key:    a word
// values: a list of documents
     Linked List locations;
     for each doc in the list of values:
         add <key, doc> to locations;
     sort locations
     emit final key-value pair (key, locations)
```

This type of relationship is shown in Figure 5, which depicts a basic representation of a multiple document inverted index, relating terms to their documents with the additional information of term frequency. When scaled to a level such as the WWW, this could be represented by each term and at which URLs that term can be found.

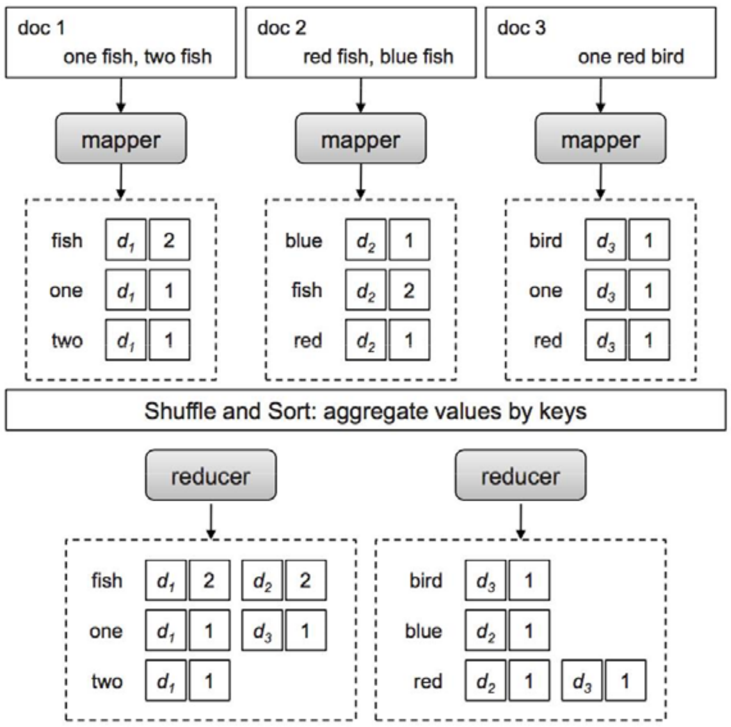


Figure 5. Basic Inverted Index <term, (docID, term frequency)>. Source: [14].

There are many advanced techniques that have been developed to enhance the content and relationship represented in an inverted index. These techniques can range from the addition of more basic information such as term frequency or term proximity to the inclusion of information regarding additional linguistic properties or even anchor text information [14]. Additionally, since creating an inverted index on the scale of the WWW results in very large output, there are a multitude of compression techniques to reduce the size of the resulting index [14], [15]. By improving the robustness of the inverted index, as well as reducing the size and complexity of the index, the eventual search capability is greatly enhanced by improving the quality and speed of the retrieved output.

3. Querying the Results

Fundamentally, the resulting output from a web search consists of the pages containing the searched for criteria. More complexly, the output is influenced by how the search algorithm scores the results of the search. Some of the simpler ways in which results can be scored are based on whether or not the pages contain some or all of the queried words, and the scoring of the results can further be enhanced by relating the proximity of those words on the page. One of the most famous ranking algorithms is PageRank [16], a method that helps to determine the importance of a webpage and which is subsequently used to boost search results. Some of the newer techniques involve identifying markup formats contained within webpages, such as Microdata, RDFa and Microformats [17], [18]. These markup formats can be used to further enhance search results. Ultimately, the boosting techniques chosen are done so with the intent of improving the search results for the intended user.

H. SUMMARY

In this chapter, an overview was provided as to why conducting an analysis of the Common Crawl dataset is a big data problem. Specifically, this is a big data problem due to the size of the dataset and that it resides in a distributed file system. Additionally, some history was provided regarding the evolution of big data analysis. In particular, this included the origins of the Google File System and MapReduce, which later became the inspiration for the Hadoop Distributed File System and Hadoop MapReduce. These tools and architecture make it possible to effectively and efficiently analyze large ad hoc datasets such as the Common Crawl archives. Finally, the framework for the creation of a robust search capability was briefly discussed.

THIS PAGE INTENTIONALLY LEFT BLANK

III. EXPERIMENTAL DESIGN

A. CAPABILITY

The goal of this project is to provide the capability to search for and extract data of interest from the Common Crawl web archives. Specifically, this capability takes the form of a word-phrase search (input) and provides relevant URLs (output). This is very similar to a standard search engine (e.g., Google, Yahoo!), except the results are scored and boosted by different criteria.

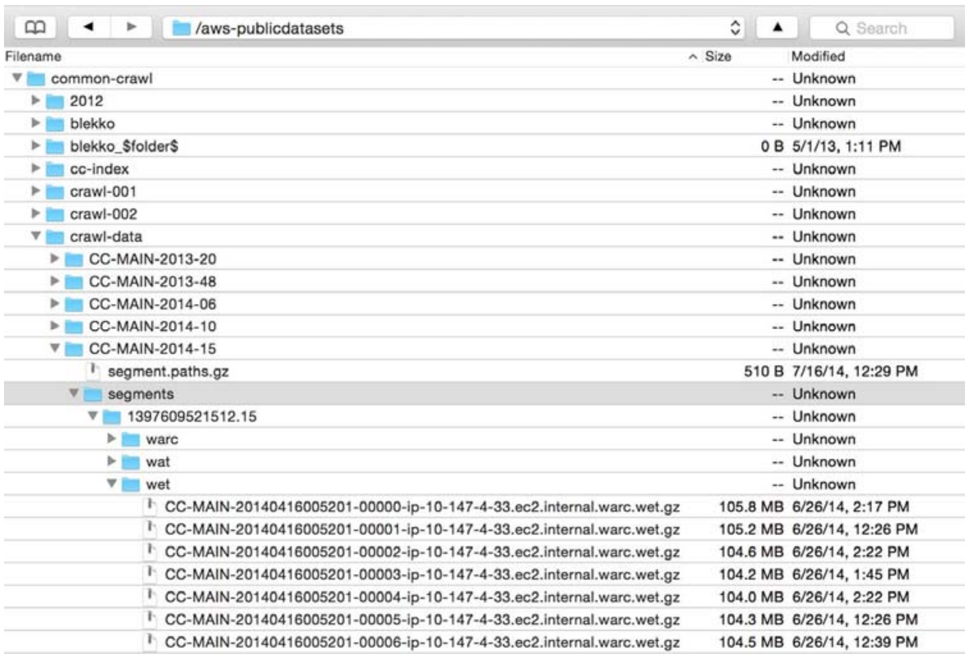
Take for example the input search phrase, *"United States of America."* The output result is a list of all URLs meeting the search and scoring criteria. Results are ordered by overall relevancy as well as other related data of interest. Relevancy basically comes down to the location and sequence of words within a record. If a record contains all the words, *"United," "States,"* and *"America,"* it is likely more relevant than a record that only contains *"United"* and *"States"* or just *"America."* Moreover, an exact match of *"United States of America"* is more relevant than the sentence, *"North America contains fifty States."* So, not only the number of words contained, but also the location of those words within the record, matter.

Other potential data of interest may also have an impact on the importance of a result. Examples of this are the geo-location of the site or the author of the site. For the first example, search results located within the United States may be of less interest than results external to the United States (e.g., Syria). This can expand into other search capabilities, such as having a search phrase and determining an author of the search results, which subsequently may result in looking for any other URLs associated with that author. Since it is neither efficient nor practical to re-perform the computation necessary for a specific search every time a new search is performed, the first step in creating a useful search capability is the generation of a robust inverted index. In order to do this, the data must be accessible and able to be read.

Since the Common Crawl data is very large and contained on AWS, the tasks of accessing, reading, and ultimately generating an index becomes more complex. In order to process crawl data, a program must be developed, written, uploaded to and run on a cluster of computers contained within the AWS. This makes sense and is practical for running a large-scale job; however, for the initial development and testing of the tools necessary to generate an index, this is a cumbersome process and can be accomplished more efficiently on a much smaller scale. In order to achieve a more efficient process, smaller chunks of data are downloaded and analyzed on a local machine in order to create the tools, which are later uploaded to AWS for large-scale analysis. Fundamentally, this process is broken down into three elements: accessing the data, developing MapReduce jobs, and running these jobs on the data.

B. ACCESSING DATA

There is a multitude of ways to access public data sets contained within AWS. Some background regarding access of this data can be obtained at http://aws.amazon.com/public-data-sets/ [12]. Files can be downloaded through various HyperText Transfer Protocol (HTTP) or S3 interface tools. Two tools utilized in the course of this thesis were S3cmd [19] and Cyberduck [20]. S3cmd is a command line S3 interface, whereas Cyberduck has a graphical interface available for interacting with S3. Within S3, individual compressed crawl files can be downloaded for local use (i.e., WARC, WET, or WAT). A typical screen shot displaying a small portion of the Common Crawl dataset file structure is shown in Figure 6. From this interface, files can easily be transferred to a local machine. Individual files of interest or entire crawls can be downloaded.



```
/aws-publicdatasets
Filename                                                                      Size   Modified
▼ common-crawl                                                                  --   Unknown
  ▶ 2012                                                                        --   Unknown
  ▶ blekko                                                                      --   Unknown
  ▶ blekko_$folder$                                                            0 B   5/1/13, 1:11 PM
  ▶ cc-index                                                                    --   Unknown
  ▶ crawl-001                                                                   --   Unknown
  ▶ crawl-002                                                                   --   Unknown
  ▼ crawl-data                                                                  --   Unknown
    ▶ CC-MAIN-2013-20                                                           --   Unknown
    ▶ CC-MAIN-2013-48                                                           --   Unknown
    ▶ CC-MAIN-2014-06                                                           --   Unknown
    ▶ CC-MAIN-2014-10                                                           --   Unknown
    ▼ CC-MAIN-2014-15                                                           --   Unknown
        segment.paths.gz                                                     510 B   7/16/14, 12:29 PM
      ▼ segments                                                                --   Unknown
        ▼ 1397609521512.15                                                      --   Unknown
          ▶ warc                                                                --   Unknown
          ▶ wat                                                                 --   Unknown
          ▼ wet                                                                 --   Unknown
              CC-MAIN-20140416005201-00000-ip-10-147-4-33.ec2.internal.warc.wet.gz   105.8 MB   6/26/14, 2:17 PM
              CC-MAIN-20140416005201-00001-ip-10-147-4-33.ec2.internal.warc.wet.gz   105.2 MB   6/26/14, 12:26 PM
              CC-MAIN-20140416005201-00002-ip-10-147-4-33.ec2.internal.warc.wet.gz   104.6 MB   6/26/14, 2:22 PM
              CC-MAIN-20140416005201-00003-ip-10-147-4-33.ec2.internal.warc.wet.gz   104.2 MB   6/26/14, 1:45 PM
              CC-MAIN-20140416005201-00004-ip-10-147-4-33.ec2.internal.warc.wet.gz   104.0 MB   6/26/14, 2:22 PM
              CC-MAIN-20140416005201-00005-ip-10-147-4-33.ec2.internal.warc.wet.gz   104.3 MB   6/26/14, 12:26 PM
              CC-MAIN-20140416005201-00006-ip-10-147-4-33.ec2.internal.warc.wet.gz   104.5 MB   6/26/14, 12:39 PM
```

Figure 6. Screen Shot of Common Crawl Data in S3

C. MAPREDUCE JOB DEVELOPMENT

As previously discussed, there are a multitude of tools that have been developed for running MapReduce jobs. For the purposes of this thesis, Java was used to develop the tools necessary to analyze the Common Crawl data. The means by which a job, written in Java, is run within the Apache Hadoop framework is in the form of a JAR file [10]. JAR is short for Java ARchive and is a single file that contains all the resources necessary to run the job. In order to accomplish this, there must be a means to write, compile and package the Java code into a JAR file as well as ensure all the proper resources are contained within that JAR file. For example, if code is written that references external libraries not contained within the standard Java and Hadoop libraries, then it needs to be included in the JAR file [10]. This is because it cannot be assumed that the ultimate machine in the cloud has those libraries available to it. Since there are different ways to specify what is ultimately contained within the JAR file, a method to include all referenced code must be used. For each of these stages, there are different tools that can be utilized.

The primary tool for Java coding utilized for this thesis was Eclipse [21]. Eclipse is a freely available Integrated Development Environment (IDE) that can be used to both write in Java as well as create the necessary JAR file to run the MapReduce job. One tool that can be utilized to properly package the necessary resources in the JAR file is called Maven [22], which is also an Apache project. There are plugins available to provide Maven functionality within Eclipse. The Maven project provides much more functionality than just building Java projects, but for the purposes of this thesis, JAR packaging is the primary use of this tool. With these tools, one can successfully write a MapReduce job in Java and create the JAR file for execution; however, there still needs to be a way to actually execute the job and assess the results.

It is possible to create the MapReduce job and run it directly on AWS for testing; however, this is a very time-consuming method to develop a MapReduce job. It is much quicker to be able to run small jobs within Hadoop locally. There are multiple ways to accomplish this, from creating your own HDFS, running a Virtual Machine (VM) with Hadoop, or even simply processing the job with a local Hadoop client. All of these methods can be accomplished through freely available sources (e.g., Hortonworks and Cloudera). For this thesis, the latter two methods were utilized. Utilizing a pre-built VM is useful in that it provides a lot of functionality with little setup and allows for exercising the job in an environment that mimics a multiple node system. Running the job through a local Hadoop client does not provide any of the features gained with the VM but is a very quick way to test a job and verify the functionality of code.

D. INDEXING DATA

In order to support a robust search capability, there first needs to be a known relationship between the words contained within crawl files and the associated URL in which those words can be found. This relationship is the inverted index. Later, this simple relationship is expanded upon to create a richer relationship; this, in turn, results in a more robust search capability. Since the WAT file only contains metadata regarding the WARC files, the files of interest in development of the inverted index are the WARC and WET files.

The WET file is the starting point for the index. This is because it contains only the text content of the associated WARC record, naturally an easier form to navigate to create the index relationship. An inverted index developed solely from the WET files results in an output that is useable for a search function; however, it does not contain additional interesting information that can only be found in the WARC file. Once the capability of creating an index from a WET file can be demonstrated, translating this functionality to a WARC file and enhancing the index is the next step.

Any record entry contained within a crawl file contains information that is not relevant or not useful in the development of an inverted index. As such, the tools created to create the index must be able to appropriately identify and traverse the data contained within the given file. Additionally, not every record entry in an archive file has to do with the content of a website crawled. Here is a fictitious example of a WET record entry associated with a website response:

```
WARC/1.0
WARC-Type: conversion
WARC-Target-URI: http://www.examplesite.com/
WARC-Date: 2015-08-05T12:27:27Z
WARC-Record-ID: <urn:uuid:23200706-de3e-3c61-a131-g65d7fd80cc1>
WARC-Refers-To: <urn:uuid:92283950-ef2f-4d72-b224-f54c6ec90bb0>
WARC-Block-Digest: sha1:XQMRY75YY42ZWC6JAT6KNXKD37F7MOEK
Content-Type: text/plain
Content-Length: 57

This is an example body extracted for a website response.
```

For a basic inverted index, there are only two useful fields contained within this example. Specifically, the useful fields are the *WARC-Target-URI*, which provides information regarding the website, and the plain text at the end of the record located after all the header information. Not only does the developed tool need to be able to identify and extract these two pieces of information, but the MapReduce job needs to be controlled in such a manner that an individual record is not broken up between different *Mappers*.

E. SUMMARY

In this chapter, the overarching concepts for the design of a robust search capability were discussed. This discussion began with exploration of the capabilities of a search tool and why having this search capability is useful. Additionally, an overview was provided on accessing the Common Crawl dataset and MapReduce job creation. The chapter was concluded with an explanation of how to create an inverted index relationship utilizing the format of the Common Crawl files.

IV. IMPLEMENTATION

A. TOOLS

Greater detail with respect to the various tools used in the development of the thesis deliverables is provided in this section. The methods provided here do not ensure or verify that all necessary supporting software has been installed on the host machine (e.g., Java).

1. Setting up a Hadoop Machine

Apache Hadoop can be installed and set up in three ways: single node setup, pseudo-distributed (cluster) setup, and fully-distributed (cluster) setup [10]. The two methods used as a part of this thesis were single node and pseudo-distributed. The purpose of setting up a Hadoop machine is to provide a method for program development and testing without running a Hadoop cluster on AWS. The two methods used in this thesis are the simplest and most effective means to accomplish this.

a. Single-Node Setup

Using a single-node setup provides the simplest means to develop, test, and debug MapReduce programs. This setup runs Hadoop as a single Java process and does not simulate a multiple node cluster [10]. In order to create this type of setup, one simply downloads and unpacks Hadoop onto the host machine. One method to accomplish this is with the following commands:

```
$ wget http://supergsego.com/apache/hadoop/common/hadoop-
       2.7.1/hadoop-2.7.1.tar.gz
$ tar xzf hadoop-2.7.1.tar.gz
```

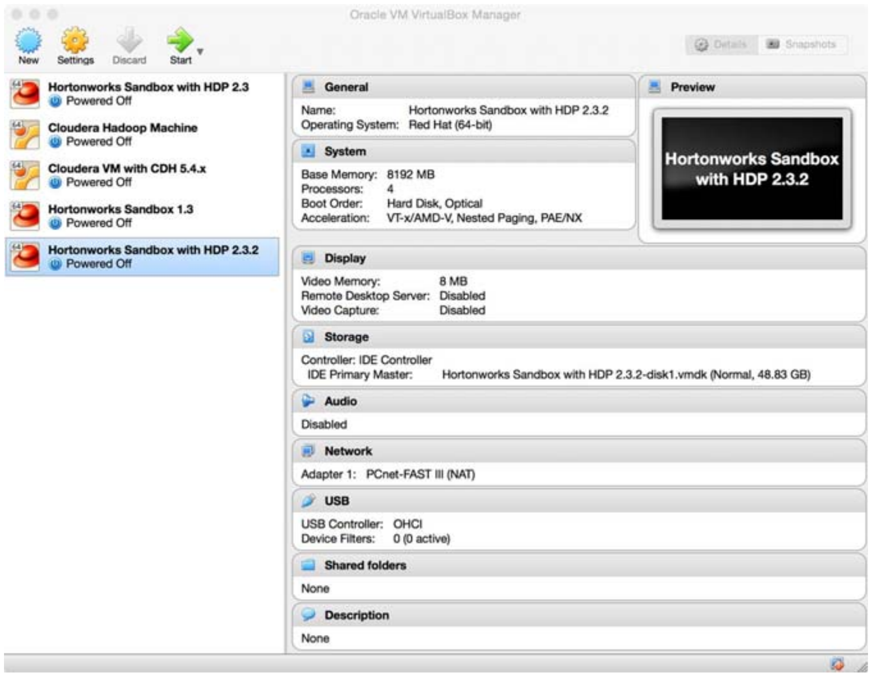
The first command (*wget*) in this sequence downloads the Hadoop source code tarball. The second command (*tar*) unpacks the specified file into its destination directory (*~/hadoop-2.7.1/*). Hadoop commands can then be directed at this location in order to run a MapReduce job on a single node. This was the primary method used in this thesis for program development and testing.

b. Pseudo-Distributed Mode Setup

A pseudo-distributed mode setup also allows for a relatively simple means to test and debug MapReduce programs. This type of setup simulates a multi-node cluster on a single machine [10]. This type of installation can be performed manually; however, there are organizations that provide freely available VMs that contain this Hadoop setup. Two of the most prominent providers of this service are Hortonworks [23] and Cloudera [24]. Both of these VMs come with installation instructions and require a VM player to be present.

2. Virtual Machine Player

The VM player used in conjunction with this thesis was VirtualBox by Oracle [25]. This VM player is freely available and provides all the necessary functionality to host either of the previously discussed Hadoop VMs. In addition to being free, VirtualBox is available for Windows, OS X, and Linux. A typical screen shot of the VirtualBox interface is illustrated in Figure 7. VirtualBox allows the user to load a number of different VMs; shown in Figure 7 are multiple different Cloudera and Hortonworks machines that were loaded for testing. The interface is the most significant difference between a Cloudera and Hortonworks VM. The Cloudera VM is an Ubuntu machine, which is a Linux operating system with a desktop interface, whereas the Hortonworks VM is a Red Hat machine that requires the user to use the command line interface (CLI) to secure shell (SSH) into the VM [23], [24]. Even though the Hortonworks VM requires a CLI, it comes installed with Hue, which gives the user a web interface to interact with some of the features of the VM. Ultimately, the user is required to utilize the CLI to run MapReduce jobs. The question of which is better, the Cloudera VM or the Hortonworks VM, is a matter of personal preference with respect to the transfer of files between the host and VM.



Hortonworks Sandbox with HDP 2.3
Powered Off
Cloudera Hadoop Machine
Powered Off
Cloudera VM with CDH 5.4.x
Powered Off
Hortonworks Sandbox 1.3
Powered Off
Hortonworks Sandbox with HDP 2.3.2
Powered Off

General
Name: Hortonworks Sandbox with HDP 2.3.2
Operating System: Red Hat (64-bit)

System
Base Memory: 8192 MB
Processors: 4
Boot Order: Hard Disk, Optical
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX

Preview
Hortonworks Sandbox with HDP 2.3.2

Display
Video Memory: 8 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage
Controller: IDE Controller
IDE Primary Master: Hortonworks Sandbox with HDP 2.3.2-disk1.vmdk (Normal, 48.83 GB)

Audio
Disabled

Network
Adapter 1: PCnet-FAST III (NAT)

USB
USB Controller: OHCI
Device Filters: 0 (0 active)

Shared folders
None

Description
None

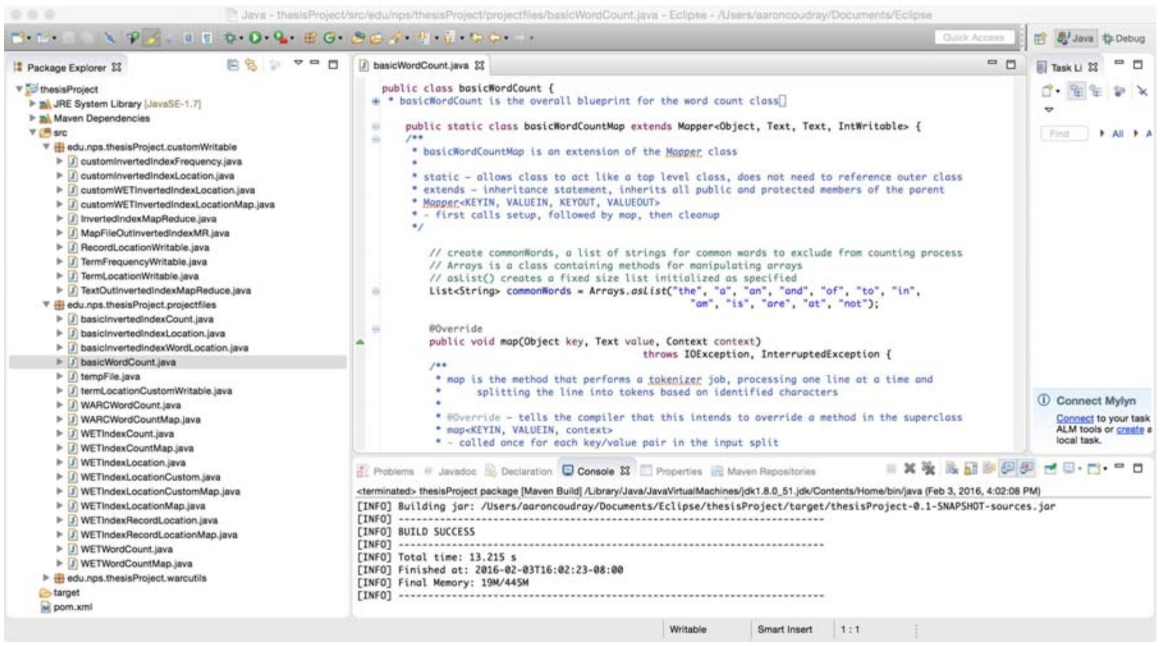
Figure 7. VirtualBox Interface

3. Java Programming

There are many tools and methods one can use to develop Java programs. With respect to this thesis, it was assumed that the reader has no prior Java programming experience; therefore, the goal was to find tools and methods that were easily accessible as well as relatively straightforward.

a. Coding

The Eclipse IDE was the primary tool utilized for code development in this thesis. The Eclipse software tool is available for Mac OS X, Windows, and Linux [21]. A typical screen shot of the Eclipse workspace is shown in Figure 8. This tool provides many helpful features, ranging from the way it displays code to the way it integrates with other libraries to the way it compiles and packages code and much more. The interfaces provided also eliminate the need for having an extensive understanding of the equivalent CLI commands.


Java - thesisProject/src/edu/nps/thesisProject/projectfiles/basicWordCount.java - Eclipse - /Users/aaroncoudray/Documents/Eclipse
Package Explorer
thesisProject
JRE System Library [JavaSE-1.7]
Maven Dependencies
src
edu.nps.thesisProject.customWritable
customInvertedIndexFrequency.java
customInvertedIndexLocation.java
customWETInvertedIndexLocation.java
customWETInvertedIndexLocationMap.java
InvertedIndexMapReduce.java
MapFileOutInvertedIndexMR.java
RecordLocationWritable.java
TermFrequencyWritable.java
TermLocationWritable.java
TextOutInvertedIndexMapReduce.java
edu.nps.thesisProject.projectfiles
basicInvertedIndexCount.java
basicInvertedIndexLocation.java
basicInvertedIndexWordLocation.java
basicWordCount.java
tempFile.java
termLocationCustomWritable.java
WARCWordCount.java
WARCWordCountMap.java
WETIndexCount.java
WETIndexCountMap.java
WETIndexLocation.java
WETIndexLocationCustom.java
WETIndexLocationCustomMap.java
WETIndexLocationMap.java
WETIndexRecordLocation.java
WETIndexRecordLocationMap.java
WETWordCount.java
WETWordCountMap.java
edu.nps.thesisProject.warcutils
target
pom.xml
basicWordCount.java
Quick Access
Java
Debug
Task List
Find
All
Connect Mylyn
Connect to your task ALM tools or create a local task.
Problems
Javadoc
Declaration
Console
Properties
Maven Repositories
Writable
Smart Insert
1 : 1

```
public class basicWordCount {
 * basicWordCount is the overall blueprint for the word count class[]

    public static class basicWordCountMap extends Mapper<Object, Text, Text, IntWritable> {
    /**
     * basicWordCountMap is an extension of the Mapper class
     *
     * static - allows class to act like a top level class, does not need to reference outer class
     * extends - inheritance statement, inherits all public and protected members of the parent
     * Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
     * - first calls setup, followed by map, then cleanup
     */

        // create commonWords, a list of strings for common words to exclude from counting process
        // Arrays is a class containing methods for manipulating arrays
        // asList() creates a fixed size list initialized as specified
        List<String> commonWords = Arrays.asList("the", "a", "an", "and", "of", "to", "in",
                                                  "am", "is", "are", "at", "not");

        @Override
        public void map(Object key, Text value, Context context)
                                        throws IOException, InterruptedException {
        /**
         * map is the method that performs a tokenizer job, processing one line at a time and
         *      splitting the line into tokens based on identified characters
         *
         * @Override - tells the compiler that this intends to override a method in the superclass
         * map<KEYIN, VALUEIN, context>
         * - called once for each key/value pair in the input split
```

```
<terminated> thesisProject package [Maven Build] /Library/Java/JavaVirtualMachines/jdk1.8.0_51.jdk/Contents/Home/bin/java (Feb 3, 2016, 4:02:08 PM)
[INFO] Building jar: /Users/aaroncoudray/Documents/Eclipse/thesisProject/target/thesisProject-0.1-SNAPSHOT-sources.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 13.215 s
[INFO] Finished at: 2016-02-03T16:02:23-08:00
[INFO] Final Memory: 19M/445M
[INFO] ------------------------------------------------------------------------
```

Figure 8. Eclipse IDE Workspace

b. Packaging

In order to run a custom MapReduce job on AWS, a custom JAR is required to be uploaded. A JAR file is essentially a way to package Java class files and other resources into a single file [26]; however, not all JAR files are created equal. Depending on the resources and libraries required for the Java program, the developer may package the JAR in such a manner that it assumes the necessary external resources are present or can be obtained by the host machine. This method provides the advantage of reducing the size of the resulting JAR file. Another way to package the JAR is to include all the necessary resources and libraries referenced by the program, such that the JAR can run regardless of whether or not those dependencies are present on the host machine. As expected, this results in a significantly larger JAR file. When creating a cluster on AWS, certain software (e.g., Pig and Hive), and hardware (e.g., cluster size and processing capability), elements of the cluster can be configured; however, it is not possible to ensure all the required dependencies are present. Thus, these dependencies must be included in the JAR file when it is packaged.

In this thesis, Maven was the primary tool used for JAR packaging. Among its other capabilities, Maven provides different methods for JAR generation and is easily integrated into Eclipse through a plugin. The ultimate goal is to create a JAR that contains all the required dependencies and resources at runtime. Some of the common terminology for this type of JAR is jar-with-dependencies, uber-jar, and fat-jar [22]. Specifics regarding the compilation and packaging of the Maven project are controlled with the pom.xml file. POM stands for project object model and, in many ways, is the blueprint of the Maven project [27]. In order to create the appropriate JAR file, the correct code must be present in the pom.xml. In this thesis, JAR packaging was accomplished with the *maven-shade-plugin*.

B. BUILDING BLOCKS

As a part of developing the tools necessary to generate an inverted index of a WARC file, incremental steps were performed to gain understanding and demonstrate capability. This primarily consisted of developing MapReduce jobs that could perform a word count function and an inverted index function on progressively more difficult file structures.

1. Components of a MapReduce Job

As previously discussed, MapReduce is the software framework utilized for processing a large dataset. This is done by splitting the dataset into many smaller chunks of data, which are then processed on separate computers. At a very basic level, this problem can be thought of as the processing of the individual chunk of data (the *Mapper*), the final conglomeration of the results of the mappers (the *Reducer*), and the overarching instructions to setup and run the job (the driver). Within a MapReduce job, there are two methods typically specified by the user: the *map* method and the *reduce* method. These methods are contained within the *Mapper* and *Reducer* classes, subsequently discussed in further detail in this section.

a. Mapper

Mapper is the overarching class that contains multiple methods associated with key-value pair generation. Overall, the purpose of the *Mapper* is to take an input key-value pair and return an intermediate key-value pair. The specific method that maps the key-value pair is the *map* method. The *map* method is the primary element that is required to be specified by the user and is the task that is performed on each of the individual chunks of data. Output from the *Mappers* is then sent to the *Reducer*, or optionally a *Combiner* that can, but is not required to, perform a similar function to the *Reducer* [8], [10].

b. Reducer

Reducer is the overarching class that contains multiple methods associated with generating the final key-value pairs. There are three phases that are commonly thought of with respect to the *Reducer*: shuffle, sort, and reduce. Shuffling and sorting have to do with the intermediate key-value pairs as output from the *Mappers*. All of these intermediate key-value pairs are grouped by key and sorted prior to becoming input to the reducing function. This takes on the form of a key with an iterable list of its values <key, list(values)>. The purpose of the *Reducer* is to take this input and reduce it to final key-value pairs. The method that accomplishes this final stage is the *reduce* method. This is another element of the MapReduce job that is typically specified by the user; however, there are some pre-built *Reducers* available in the Hadoop libraries [8], [10].

c. The Driver

The overall driver method for the MapReduce job is responsible for setting up the environment for the job and the execution of the job. There are multiple ways to handle potential arguments to the job and establish the job parameters. At the most basic level, this method must create the job: configure the *Mapper* and *Reducer*; configure the HDFS input and output file paths; configure the JAR class; and finally, submit the job [8], [10].

2. Word Count with Text Files

The word count job is often equated to the hello world example for MapReduce programming. As such, this was the first building block developed with the primary purpose of demonstrating the ability to create and run a MapReduce job. The basic premise of this example is to be able to read in a set of text files and output all words contained within those files and the associated number of times the word occurs. The associated code for this function is contained within Appendix A.

In this example, the dataset is broken up by line of text. Each line of text is sent to a map task for processing. Within each *Mapper*, each line of text is tokenized into individual words, and then each word is checked against a list of common words. For each word not contained in the common words list, that word is emitted from the *Mapper* as a part of an intermediate key-value pair. Since the overall function is to perform a word count, that word is emitted with a count of one <term, 1>. Here is a simple example for demonstration purposes (not considering a list of common words):

```
// example phrase:
See Spot run, run Spot run.

// the end result of the Mappers will be:
< see, 1>
<spot, 1>
< run, 1>
< run, 1>
<spot, 1>
< run, 1>
```

Each of these pairs is considered an intermediate key-value pair and is subsequently sent to the *Reducer* where it is shuffled, sorted, and reduced.

The *Reducer* then performs the function of creating the final count for each word. Continuing with the previous example, we see that the output of the shuffle and sort phase results in the following:

```
// results from shuffle and sort phase
< see, (1)>
<spot, (1, 1)>
< run, (1, 1, 1)>
```

Once the intermediate key-value pairs have been shuffled and sorted, the *Reducer* performs its function. In the case of a word count, the function of the *Reducer* is to add up the total number of instances of each key (term) and output a final key-value pair in the form of <term, sum>. Continuing the example, we see that this results in:

```
// final <key, value> pairs
< see, 1>
<spot, 2>
< run, 3>
```

The final output is a file, or files depending on the number of *Reducers*, that contains all of the final key-value pairs.

3. Inverted Index with Text Files

The next building block was to create a MapReduce job that created a basic inverted index. This example is similar to the previous word count example in that it also takes a set of text files as its input. The output is different in that it creates a relationship between the words contained in all the files and the filenames in which those same words are located. To demonstrate the capability to add additional information to the index relationship, this example also outputs the word count for each word with respect to each individual text file. The associated code for this function is contained within Appendix B.

As in the word count example, the dataset for this process is broken up by line of text from each individual file. Each line of text is sent to a map task for processing. As before, each *Mapper* tokenizes the line of text into each individual word and checks each word against a list of common words. In this instance, if the word is not filtered by the common words list, the word is then appended with the filename in which it came from <word>@<filename>. This appended value becomes the intermediate key and is emitted from the *Mapper* with a count value of one <word@filename, 1> to support the subsequent count determination for each file. Here is a simple example for demonstration purposes (not considering a list of common words):

```
// example input from two files:
// filename: file1.txt
// contents:
See Spot run, run Spot run.

// filename: file2.txt
// contents:
Look at Spot run.

// the end result of the Mappers will be:
< see@file1.txt, 1>
<spot@file1.txt, 1>
< run@file1.txt, 1>
< run@file1.txt, 1>
<spot@file1.txt, 1>
< run@file1.txt, 1>
<look@file2.txt, 1>
<  at@file2.txt, 1>
<spot@file2.txt, 1>
< run@file2.txt, 1>
```

As before, each of these intermediate key-value pairs are then shuffled and sorted prior to being reduced by the *Reducer*. For the provided example, this phase results in the following output:

```
// results from shuffle and sort phase:
<  at@file2.txt, (1)>
<look@file2.txt, (1)>
< see@file1.txt, (1)>
<spot@file1.txt, (1, 1)>
<spot@file2.txt, (1)>
< run@file1.txt, (1, 1, 1)>
< run@file2.txt, (1)>
```

Once the output has been shuffled and sorted, the *Reducer* sums up the count of each term from its respective file and outputs a final key-value pair in the form of <term@filename, sum>. Finishing the example, we see that this results in:

```
// final <key, value> pairs:
<  at@file2.txt, 1>
<look@file2.txt, 1>
< see@file1.txt, 1>
<spot@file1.txt, 2>
<spot@file2.txt, 1>
< run@file1.txt, 3>
< run@file2.txt, 1>
```

The final output is a file, or files depending on the number of *Reducers*, that contains all of the final key-value pairs.

C. APPLICATION TO WET FILES

The next building block was to take the concepts of the word count and inverted index and apply them to a WET file. At this stage, the problem became significantly more complex due to the structure and contents of a WET file as compared to a text file. When dealing with text files, we assumed that each term present in the file was important to account for, as well as that each individual file was associated with an individual location. When dealing with the WET files, these assumptions were no longer valid.

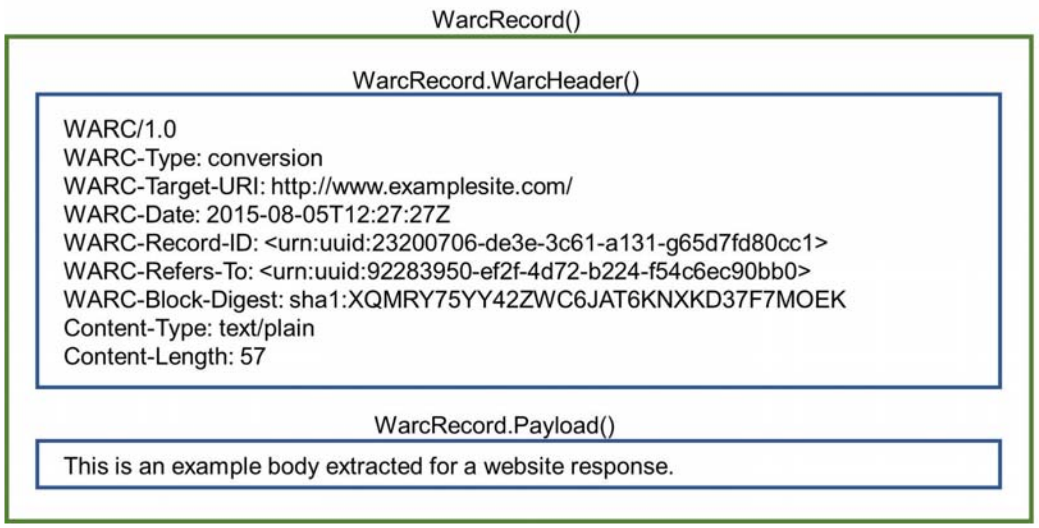
Numerous records associated with different website responses are contained within an individual WET. Additionally, each record has an associated header which provides information regarding the origin of the record; therefore, as a part of processing each record within a WET file, it is important to be able to associate each header with its respective payload but at the same time distinguish between them when processing the contents. This type of processing also requires that individual records not be broken up between *Mappers*. When processing the text files, it is acceptable to break up data chunks by individual lines of text from the files; however, if this type of data distribution occurred within the WET files, it results in a dissociation of header and payload information. Therefore, the MapReduce program must ensure the continuity of individual records by forcing the smallest data chunk allowable to be an individual record.

1. Word Count with a WET File

As previously discussed, there are two main characteristics of the WET file that are important in how the file was processed: the ability to maintain the integrity of an individual record entry and the ability to distinguish between the header and payload content. Since there is already research being conducted with the Common Crawl dataset, there are some tools available for processing of WARC files within the structure of a MapReduce program. Two such tools are the Webarchive Commons repository [28] and the Java Web Archive Toolkit (JWAT) repository [29]. Both of these tools have features that can be employed to limit the smallest data chunk to that of an individual record entry as it is sent to various *Mappers*. Additionally, each of these libraries provide a multitude of tools that can be used for data processing. The JWAT repository for WET file processing was utilized in this thesis.

The JWAT libraries, as a whole, provide tools for reading and writing WARC, ARC, and GZIP files [29]. The specific tools of interest within the JWAT libraries are those that allow for reading and processing of the WARC file. Even though a WARC file is significantly different than a WET file, the overall structure of header and payload are consistent enough to process with the same tools. The difference is demonstrated in the specific processing of the header and payload contents.

When a record is read in through the JWAT libraries, it creates an object from the class *WarcRecord*. The *WarcRecord* class of objects is the object that contains the *header* and *payload* fields. The *header* field is another object from the class of *WarcHeader*, which has fields associated with all possible named fields as specified in the WARC ISO. Each of these named fields is automatically assigned to the object fields when processed by the JWAT libraries. The *payload* object contains the remainder of the record entry, outside of the header information, and has no other specific fields associated with its contents. A brief example depicting how an individual record is broken up is shown in Figure 9 [29].



```
WARC/1.0
WARC-Type: conversion
WARC-Target-URI: http://www.examplesite.com/
WARC-Date: 2015-08-05T12:27:27Z
WARC-Record-ID: <urn:uuid:23200706-de3e-3c61-a131-g65d7fd80cc1>
WARC-Refers-To: <urn:uuid:92283950-ef2f-4d72-b224-f54c6ec90bb0>
WARC-Block-Digest: sha1:XQMRY75YY42ZWC6JAT6KNXKD37F7MOEK
Content-Type: text/plain
Content-Length: 57
```

This is an example body extracted for a website response.

Figure 9. Example WET File Entry Identifying JWAT WARC Record Objects

In order to access individual fields associated with a header, those fields must simply be referenced from the respective variable. The following pseudo-code demonstrates how to access fields within an individual *WarcRecord*:

```
// object containing the entire record
// value: the WarcRecord object as read in by the mapper
WarcRecord record = value;

// object containing the record header
WarcHeader recordHeader = record.getHeader();

// object containing the record payload
Payload recordPayload = record.getPayload();

// variable containing the target url field
String recordURL = record.header.warcTargetUriStr;
     // OR
String recordURL = recordHeader.warcTargetUriStr;
```

These features facilitate navigating and processing the broader fields contained within each individual record entry; however, the *payload* must still be processed in more of a brute force method. Similar to the word count for text files, this was accomplished using a tokenizer. In later portions of this thesis, a technique utilizing regex (regular expressions) is used. The associated code for this function is contained within Appendix C.

Regular expressions are used for pattern matching. This technique is not unique to Java and can be found in a multitude of different programming languages. In Java the regex process primarily involves the use of two classes: *Pattern* and *Matcher*. The *Pattern* class is used to define the string search pattern. The *Matcher* class is then used to find the specified pattern within the input sequence. Overall, the regex technique for pattern matching is a much more versatile and robust method for analyzing character sequences than a tokenizing method [30].

Processing of the record *payload* is nearly identical to that as in the first word count example for text processing. The only difference is that entire *payload* is processed and tokenized at a single *Mapper* rather than an individual line of text as in the first word count example. This results in the entire *payload* being broken up into individual words, which are then checked against a list of common words. If the word is not filtered by the common words list, it is emitted from the *Mapper* as a part of an intermediate key-value pair in the form of <term, 1>. Here is a simple example containing two records (oversimplified) for demonstration purposes (not considering a list of common words):

```
// first example record:
WARC-Target-URI: http://www.firstexamplesite.com/

See Spot run.

// second example record:
WARC-Target-URI: http://www.secondexamplesite.com/

Run Spot run.

// the end result of the Mappers will be:
< see, 1>
<spot, 1>
< run, 1>
< run, 1>
<spot, 1>
< run, 1>
```

In this problem, it is not important which records the words came from since the end result is an overall word count of all words. As before, each of these pairs is considered an intermediate key-value pair and is sent to the *Reducer* where it is shuffled, sorted, and reduced.

The *Reducer* then performs the function of creating the final count for each word. Continuing with the previous example, we see that the output of the shuffle and sort phase results in the following:

```
// results from shuffle and sort phase
< see, (1)>
<spot, (1, 1)>
< run, (1, 1, 1)>
```

Once the intermediate key-value pairs have been shuffled and sorted, the *Reducer* then performs its function. In the case of a word count, the function of the *Reducer* is to add up the total number of instances of each key (term) and output a final key-value pair in the form of <term, sum>. Continuing the example, we see that this results in:

```
// final <key, value> pairs
< see, 1>
<spot, 2>
< run, 3>
```

The final output is a file, or files depending on the number of *Reducers*, that contains all of the final key-value pairs.

2. Inverted Index with a WET File

As with the progression with text file processing, the next step in processing the WET files was to create a MapReduce job that created an inverted index. The end goal was to create a relationship between each word contained within the records, the URL of the respective record, and the relative location of the word in the payload of the respective record. Each of these pieces of information was important to capture in support of creating a relationship that can be utilized in a search. Take the example of the phrase "*United States of America,*" shown in Figure 10.

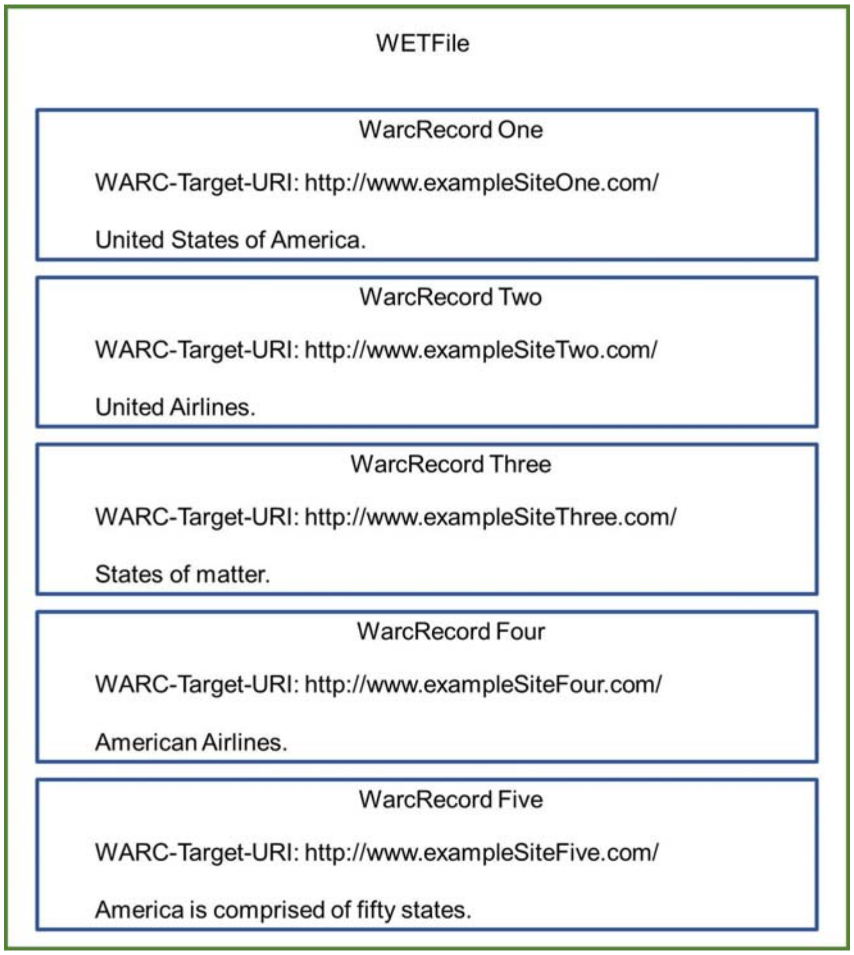

WETFile
WarcRecord One
WARC-Target-URI: http://www.exampleSiteOne.com/
United States of America.
WarcRecord Two
WARC-Target-URI: http://www.exampleSiteTwo.com/
United Airlines.
WarcRecord Three
WARC-Target-URI: http://www.exampleSiteThree.com/
States of matter.
WarcRecord Four
WARC-Target-URI: http://www.exampleSiteFour.com/
American Airlines.
WarcRecord Five
WARC-Target-URI: http://www.exampleSiteFive.com/
America is comprised of fifty states.

Figure 10. Example WET File with Five Record Entries

It is no longer sufficient to have the word location relationship contain the file of origin, since each WET file contains numerous records from different websites. Finding "*United*," "*States*," and "*America*" on different sites is not nearly as relevant as finding them all located on the same site. The additional information of word relative position can then help to identify the word relationships: are they used continuously in a phrase or are they not? This was accomplished by creating a custom tuple-data type, which contained the word, URL, and relative location in a single variable. The associated code for the custom data type is contained in Appendix D, and the code for this function is contained in Appendix E.

As in the WET file word count example, the dataset for this process is broken up by record; however, unlike previous examples, this process utilizes a regex pattern matching technique to parse the record *payload*. Once patterns are identified, they are checked against a list of common words. If the word is not filtered, it is then placed into the custom data type with its related location information in the form of <word, (URL, location)>. Here is a simple example for demonstration purposes (not considering a list of common words):

```
// first example record:
WARC-Target-URI: http://www.firstexamplesite.com/

See Spot run, run Spot run.

// second example record:
WARC-Target-URI: http://www.secondexamplesite.com/

Look at Spot run.

// the end result of the Mappers will be:
< see,  (http://www.firstexamplesite.com/, 1)>
<spot,  (http://www.firstexamplesite.com/, 2)>
< run,  (http://www.firstexamplesite.com/, 3)>
< run,  (http://www.firstexamplesite.com/, 4)>
<spot,  (http://www.firstexamplesite.com/, 5)>
< run,  (http://www.firstexamplesite.com/, 6)>
<look, (http://www.secondexamplesite.com/, 1)>
<  at, (http://www.secondexamplesite.com/, 2)>
<spot, (http://www.secondexamplesite.com/, 3)>
< run, (http://www.secondexamplesite.com/, 4)>
```

As before, each of these intermediate key-value pairs are shuffled and sorted prior to being reduced by the *Reducer*. For the provided example, this phase results in the following output:

```
// results from shuffle and sort phase:
<  at, (http://www.secondexamplesite.com/, 2)>
<look, (http://www.secondexamplesite.com/, 1)>
< see,  (http://www.firstexamplesite.com/, 1)>
<spot, [(http://www.firstexamplesite.com/, 2),
         (http://www.firstexamplesite.com/, 5),
        (http://www.secondexamplesite.com/, 3)]>
< run, [(http://www.firstexamplesite.com/, 3),
         (http://www.firstexamplesite.com/, 4),
         (http://www.firstexamplesite.com/, 6),
        (http://www.secondexamplesite.com/, 4)]>
```

Once the output has been shuffled and sorted, the *Reducer* consolidates the values for each key so that each contains unique URLs with multiple locations. This is accomplished using a hash map and results in a final key-value pair in the form of <term, {[URL, list(locations)], URL, list(locations), …}>. Finishing the example, we see that this results in:

```
// final <key, value> pairs:
<  at, (http://www.secondexamplesite.com/, 2)>
<look, (http://www.secondexamplesite.com/, 1)>
< see,  (http://www.firstexamplesite.com/, 1)>
<spot, [(http://www.firstexamplesite.com/, 2, 5),
        (http://www.secondexamplesite.com/, 3)]>
< run, [(http://www.firstexamplesite.com/, 3, 4, 6),
        (http://www.secondexamplesite.com/, 4)]>
```

The final output is a file or files, depending on the number of *Reducers*, containing all of the final key-value pairs.

D. APPLICATION TO WARC FILES

The final phase in this approach was the development of the tools and techniques necessary to create the same tools (i.e., word count and inverted index) for the WARC file. While many of the concepts utilized for processing the WET files can be directly applied to the WARC file, there are many additional elements contained within a WARC file that require consideration.

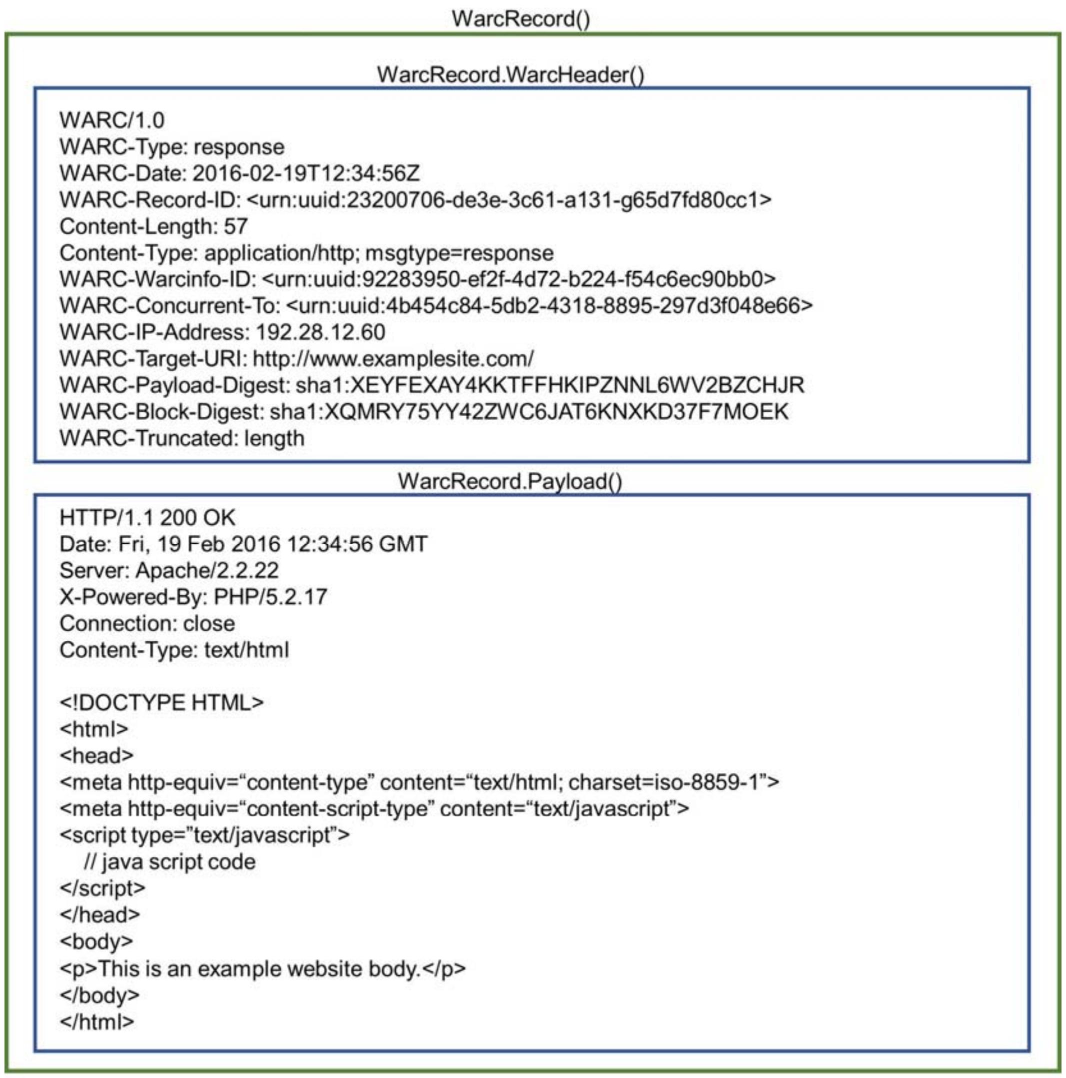
There are two overarching issues that need to be addressed in order to effectively process the WARC files. The first is that the WARC file contains multiple types of records, (e.g., *response*, *request*, and *metadata*) beyond that of the record-of-interest (i.e., *response*) [3]. The second issue is that the *payload* for the records-of-interest are no longer plain text; they contain the complete scheme-specific response to include network protocol. For example, the response from a website following an HTTP scheme contains all elements as defined by the HTTP standard (RFC 2616) [31]. The issue of record type identification was handled with the JWAT tools introduced for processing of WET files; however, handling of the *payload* required different techniques that had, up to this point, not been explored.

Websites can be comprised of a multitude of different schemes (e.g., HTML, XML, Compressed Files). The format of the scheme-specific response is different, and for as many different types of sites/files that are crawled, there are as many different responses obtained. Because each of these categories of responses must be handled (i.e., identified and parsed) differently, the focus of this thesis is on creating the capability to handle HTML-type responses.

1. Word Count with a WARC File

As discussed, the first capability difference in processing a WARC file versus a WET file is to be able to identify the *response* entries. An example *response* WARC record entry is shown in Figure 11. Utilizing the same JWAT libraries as before, we can extract the *WARC-Type* field from the record header. This is the header field that identifies which type of record entry follows. The following pseudo-code then shows how the records-of-interest can be identified for processing:

```
Mapper (LongWritable key, WarcRecord value):
// key:   not used
// value: the warc record entry
     if value.header.warcTypeStr equals response
         // process record
     else
         // go to next record
```



```
WARC/1.0
WARC-Type: response
WARC-Date: 2016-02-19T12:34:56Z
WARC-Record-ID: <urn:uuid:23200706-de3e-3c61-a131-g65d7fd80cc1>
Content-Length: 57
Content-Type: application/http; msgtype=response
WARC-Warcinfo-ID: <urn:uuid:92283950-ef2f-4d72-b224-f54c6ec90bb0>
WARC-Concurrent-To: <urn:uuid:4b454c84-5db2-4318-8895-297d3f048e66>
WARC-IP-Address: 192.28.12.60
WARC-Target-URI: http://www.examplesite.com/
WARC-Payload-Digest: sha1:XEYFEXAY4KKTFFHKIPZNNL6WV2BZCHJR
WARC-Block-Digest: sha1:XQMRY75YY42ZWC6JAT6KNXKD37F7MOEK
WARC-Truncated: length
```

```
HTTP/1.1 200 OK
Date: Fri, 19 Feb 2016 12:34:56 GMT
Server: Apache/2.2.22
X-Powered-By: PHP/5.2.17
Connection: close
Content-Type: text/html

<!DOCTYPE HTML>
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=iso-8859-1">
<meta http-equiv="content-script-type" content="text/javascript">
<script type="text/javascript">
   // java script code
</script>
</head>
<body>
<p>This is an example website body.</p>
</body>
</html>
```

Figure 11. Example WARC File Entry Identifying JWAT WARC Record Objects

Once the proper records are identified, the extraction of the *payload* is performed in the same manner as for the WET files. The next step is identifying whether or not the *payload* contained HTML and, if so, the subsequent extraction of the target text from within the HTML payload. The associated code for this function is contained within Appendix F.

The primary tool utilized to parse through the HTML payload was jsoup, which is a Java library for HTML parsing [32]. The jsoup library provides tools to perform tasks such as parsing, scraping, and extracting various attributes from HTML. This library was successfully utilized to both identify HTML content and extract the text body of the page. Each of these functions was accomplished with different jsoup tools; the more complex task being the identification of HTML.

RFC 2616 defines certain fields that should be contained within a HTTP response header. One of these fields is called *Content-Type*, which defines the media type of the response [31]. Utilizing jsoup, we extracted this content field and checked for the keyword *html* in order to assess whether or not the content body was HTML. This was accomplished by a *select* function that was used to perform a Cascading Style Sheets (CSS) query for the appropriate meta tag in the content. If the correct meta tag was found, it was extracted into a string and checked against *html* as a substring. If this check was positive, the process continued to extract the text from the *payload*; otherwise, it was assumed the checked record did not contain HTML and was subsequently skipped. It was later determined that some HTML records were missed utilizing this method and another approach was investigated. This technique is still useful in that it demonstrates the process that would likely be utilized in order to further extract fields-of-interest within HTML code.

Rather than look for specific meta tags associated with HTML content, it seemed reasonable to attempt to identify the <html></html> tags within the record. These tags mark the beginning and end of the HTML structure and are present regardless of the specific character set or content type utilized by the code. In order to do this, the *select* function was again utilized. This time the CSS query was set to look for the <html> tag. If this tag is found, it returns the entire content, including the specified tag; however, if the <html> tag is not found it returns *null*. This allowed for a simple comparison against *null* to determine the presence of an HTML record. In principle, this method was expected to identify more HTML records than the method triggering off of the *Content-Type* meta tag. A simple analysis on a random WARC file was performed to provide a basic comparison of these two methods. The results of this analysis are shown in Table 1.

Table 1. Comparison of HTML Identification Methods

Record Feature	No.		Record Feature	No.
Contain “Content-Type” Tag	32,210		Contain “html” in Tag	32,131
No “Content-Type” Tag	21,292		Contain <html> Tag	32,210
Total Entries	53,502			

Within the WARC file there were 53,502 *response* records. The first analysis was to determine how many of the *response* records contained the *Content-Type* meta tag and, subsequently, contained the string “html.” The second analysis was to determine how many of the *response* records contained the <html> tag. Not shown in this data is the fact that within this one WARC file there are over two hundred different values found within the *Content-Type* meta tag. In conclusion, both methods were found to capture all or almost all of the HTML *response* records, with the method identifying the <html> tag performing slightly better.

If the record payload was identified as an HTML response, it was processed for content in support of the word count function. In order to extract this content, a jsoup function was utilized that accesses the contents contained within the *body* element of the HTML. Once the payload body was obtained, it could be processed in a manner similar to previous examples. For this function, the method of pattern matching through use of regex was utilized to identify distinct words. Once words had been identified, they were passed through the process of checking for common words and then being emitted from the *Mapper* as an intermediate key-value pair. Upon emission, they were sent to the *Reducer* for the shuffle, sort, and reduce process.

Lastly, the *Reducer* performed the function of creating the final count for each word. As with each word count function, this meant the *Reducer* added up the total number of instances of each key (term) and output a final key-value pair in the form of <term, sum>. The final output is a file, or files depending on the number of *Reducers*, that contains all of the final key-value pairs.

2. Inverted Index with a WARC File

As with previous examples, the next step in processing the WARC files was to develop the MapReduce job that created an inverted index. Similar to that for the WET files, this inverted index built the relationship between each word contained within a WARC record, the associated URL of that record, and, finally, the relative location of the word within the *payload* of that record. This implementation also utilized the custom tuple-data type, and its associated code can be found in Appendix G.

This tool essentially resulted in a combination of the WET file inverted index and WARC file word count tools. The content and forming of the index is nearly identical to that developed for the WET files, that being the use of a tuple-data type to group the term, URL, and relative location. The key difference is in how the *payload* was parsed in order to determine which terms were present. Parsing of the *payload* was nearly identical to the methods developed in the WARC word count function. The WARC inverted index tool is essentially formed by combining two techniques: the inverted index relationship using a custom data type and parsing the WARC payload.

As in the WARC file word count example, the dataset for this process is broken up by record. Additionally, this process utilized a regex pattern matching technique to parse the record *payload*. Once patterns are identified, they are checked against a list of common words. If the word is not filtered, it is then placed into the custom data type with its related location information in the form of <word, (URL, location)>. Each of these intermediate key-value pairs are then shuffled, sorted, and reduced in a manner similar to previous examples. Ultimately, the *Reducer* performs the consolidation of the records so that each key contains unique URLs with multiple locations and outputs the final key-value pair to a file or files. The structure of the WARC inverted index is the same as the pseudo-code examples provided during the discussion of the WET file inverted index.

E. SUMMARY

The tools used in the development of the deliverables of this research were presented in this chapter. This included discussion of the various tools created in the development of a MapReduce job capable of generating an inverted index from a WARC file. Specifically, this included jobs capable of performing a word count and creating an inverted index on a text file and jobs capable of performing a word count and creating an inverted index from a WET file. This culminated with the tools capable of performing a word count and creating an inverted index on a WARC file. Selected code developed as a part of this thesis has been included within the Appendices, with guidance for loading source code and generating JAR files contained within Appendix H and I, respectively. The complete project files are available in the Supplemental Material (Appendix K).

V. RESULTS

Testing was conducted throughout the design and implementation process for this work. Each of the building blocks discussed in Chapter IV was tested in a local single node Hadoop machine as well as within AWS. This process was used to not only verify operation but also to identify issues and improve overall functionality. This culminated with the evaluation of the job designed to generate an inverted index from WARC file(s).

A. MAPREDUCE JOB ON A SINGLE NODE (LOCAL)

As previously discussed, speed is the primary benefit of running a local single node Hadoop machine. It is possible to perform multiple iterations—alter code, create an updated JAR file, run that job, and observe the results—in the same period it takes to run one equivalent job in AWS. This makes local testing a much more efficient way to locate and correct errors as well as to improve functionality.

The first objective was to obtain a MapReduce job that successfully created an inverted index utilizing a single WARC file as an input. This demonstrated three things: (1) that the WARC file could be properly read and parsed in a MapReduce job; (2) that the inverted index relationship between term and URL could be extracted; and (3) that the additional information of term relative position could be captured and incorporated into the index. The input was controlled as a single WARC file in an input directory, and the output was controlled with a single *Reducer*, resulting in a single output file. The following is a small excerpt from the output:

```
monteray : http://www.tripadvisor.nl/SmartDeals-g51891-
Grants_Pass_Oregon-Hotel-Deals.html:853,
http://www.ratebeer.com/maps/print-8702.htm:82,

monterblanc : https://ms.wikipedia.org/wiki/Saint-Nicolas-
du-Tertre:437,

monterchi : http://saterdesign.com/product-
category/concrete-home-
plans/?dir=desc&mainlevelmaster=184&order=plantotal_living_
sqft&planbathrooms=102&plangaragedoors=189:381,

montere : https://da.wikipedia.org/wiki/Westwall:2476,
monterescu : http://www.haaretz.com/misc/comment-page/eh-
makes-no-sense-the-military-doesnt-rule-after-the-election-
19.2142582:1212,  http://www.haaretz.com/misc/comment-
page/quot-patricks-brother-quot-what-plain-quot-facts-quot-
would-they-be-19.1046099:1206,
http://www.haaretz.com/misc/comment-page/population-growth-
stops-at-6-billion-19.1213118:1200,

monteret : https://da.wikipedia.org/wiki/Westwall:2251,

monterey : http://lubbockonline.com/filed-online/2014-05-
03/region-i-high-school-baseball-playoff-pairings:421, 416,
419,
http://www.groworganic.com/fertilizers.html?garden_size=529
&limit=120&organic_use=514&solution_fertilizer=349:2217,
2038,  http://www.snagajob.com/job-search/j-
retail+sales+consultant+monterey+ca/w-93955/q-atandt:176,
http://www.silentpcreview.com/forums/viewtopic.php?t=66667p
=579171:778, 1224, 2372,
http://streema.com/users/sofivanmensvoort:439,
http://www.spokeo.com/Harry-Eldridge:1260,
https://www.groupon.com/local/san-diego-country-estates-
ca/wineries:443,
http://cars.automotive.com/toyota/highlander/2007/compare/t
rims/t3-12-4/:530,
```

The last entry in the previous example is for the term *monterey*. This entry has been colored to facilitate identifying the different inverted index relationships. Each URL that contained *monterey* within its text is shown in a different color, for a total of eight different URLs in the single WARC file used as input. The term and locations are separated by a colon (:), as well as the additional information of the term relative location (underlined). For instance, the first entry of *lubbokonline.com* contains the term *monterey* at three different locations: the 421st, 416th, and 419th position. This demonstrated that the MapReduce job successfully accomplished its purpose on a single file.e These results also showed that the colon may not have been the best choice in delimiter, since it is often contained in URLs; it was subsequently changed to the vertical slash (|).

The next step was to test the ability to index multiple input files from the same directory as well as from different directories. A single directory with multiple files is equivalent to this structure:

```
../input---warcFile1.warc.gz
         |
         |-warcFile2.warc.gz
         |
         |-warcFile2.warc.gz
```

Additionally, the job needed to be able to differentiate between WARC, WET, and WAT files since they are all contained within the Common Crawl dataset. A file structure containing one or more subdirectories with multiple files is equivalent to this structure:

```
../input---segment1---warcFile1.warc.gz
         |          |
         |          --wetFile1.warc.wet.gz
         |          |
         |          --watFile1.warc.wat.gz
         |
         |-segment2---warcFile2.warc.gz
                    |
                    --wetFile2.warc.wet.gz
                    |
                    --watFile2.warc.wat.gz
```

It is possible to specify multiple input paths when creating a MapReduce job, but this becomes very cumbersome and unrealistic when expanded to the scale of analyzing hundreds of WARC files. As such, there are two common ways to handle this problem: (1) path filtering and (2) globbing [8]. Path filtering is a more robust technique that essentially examines entire file paths and can be matched using the regex technique in order to identify the desired files to input. Globbing is a simpler technique that has some basic matching capability, including wildcards, (i.e., the (*) character), to search file paths. This technique is not as flexible as path filtering and works better if the file structure is uniform, such as in the example provided. Because the Common Crawl dataset is well structured within a given crawl, the technique of globbing was tested and used.

In order to test the globbing capability, a similar multiple directory structure was created locally with multiple WARC files and other non-WARC type files. These additional files were present to ensure the specified glob could properly identify the desired files. The appropriate input path glob to analyze all WARC files for the previous example is:

```
input/*/*.warc.gz
```

This glob searches any folder one level deep into the input folder for any file with the extension of *warc.gz*. Ultimately, the MapReduce job was successful in handling multiple WARC files contained in separate subdirectories. One key finding from this testing was that the resulting output files grew rapidly in size and presented problems on multiple levels. The first level of concern was the performance of the MapReduce job, specifically, how many *Reducers* to use. Initially, this job only utilized one *Reducer*, which not only becomes a bottleneck for task processing but also becomes related to the issue of memory. As the number of input files grows, the amount of memory utilized grows, and this is particularly a concern for the *Reducers*. The *Mappers* are processing relatively smaller chunks of data, but the resulting intermediate key-value pairs increase in number and size for processing at the *Reducers*. Not knowing how this job would perform on AWS, only the issue regarding the number of *Reducers* was addressed. Specifically, the number of *Reducers* was increased.

By increasing the number of *Reducers*, we effectively split the intermediate key-value pair space in size by the number of *Reducers*. There are methods and rules of thumb to help determine the appropriate number of map and reduce tasks based on the job being performed [10]. The intent of these methods is not necessarily to execute the job in the fastest way possible but to exercise the most efficient use of computing power and resources. These methods were not explored as a part of this thesis. One side effect of having multiple *Reducers* is that each reducer creates its own output file. There is no particular coordination between the *Reducers* to organize intermediate key-value pairs. This can be specified through the use of a *Partitioner* [10].

For ease of analysis, it was decided to partition the intermediate key-value pairs such that the resulting output was organized by the first character of each term. This resulted in specifying 37 partitions, one for each number and character in the English alphabet and one catch-all partition. This, in turn, meant that 37 *Reducers* were required, one for each partition. The resulting output files for the MapReduce job were then organized by first character, and within each output file, the terms were sorted alphabetically. This updated job was then performed on a different individual WARC file. A summary of output file sizes for a single WARC file is shown in Table 2. This data is the result of processing a single WARC file with an input file size of 887.5 MB. The size of the output files comes to a total of 1.99 gigabytes (GB), with the largest individual output file at 184.4 MB.

Table 2. Inverted Index Output File Size from Processing a Single WARC File

File	Character	Size (MB)		File	Character	Size (MB)
00000	0	29.6		00019	j	26.7
00001	1	54.0		00020	k	24.2
00002	2	56.4		00021	l	72.5
00003	3	23.3		00022	m	101.4
00004	4	19.2		00023	n	55.7
00005	5	18.7		00024	o	56.3
00006	6	12.5		00025	p	131.4
00007	7	11.4		00026	q	12.1
00008	8	13.9		00027	r	88.1
00009	9	11.8		00028	s	184.4
00010	a	11.0		00029	t	104.5
00011	b	86.4		00030	u	40.1
00012	c	172.4		00031	v	37.6
00013	d	84.8		00032	w	57.2
00014	e	72.6		00033	x	10.7
00015	f	85.3		00034	y	20.7
00016	g	57.8		00035	z	9.3
00017	h	72.4		00036	(other)	3.3
00018	i	55.6				

Part of the large difference between the resulting total output file size and input file size is that the resulting output file is not compressed, whereas the input files are compressed. Specifically, in the form of a GZIP compressed file. Prior to addressing any of these concerns, it was important to first test the MapReduce job on AWS to verify that it worked in a distributed system by accessing the Common Crawl files.

B. MAPREDUCE JOB ON ELASTIC MAP REDUCE (AWS)

Amazon Elastic MapReduce (EMR) is the primary tool for running MapReduce jobs within AWS. The EMR service provides coordination between S3 and EC2 in order to create a Hadoop cluster (see Figure 12). Scalable EC2 instances are created as computational nodes, while S3 is used for input and output storage as well as for loading the custom MapReduce job. A more detailed walkthrough of running a job within AWS is provided in Appendix J [33].

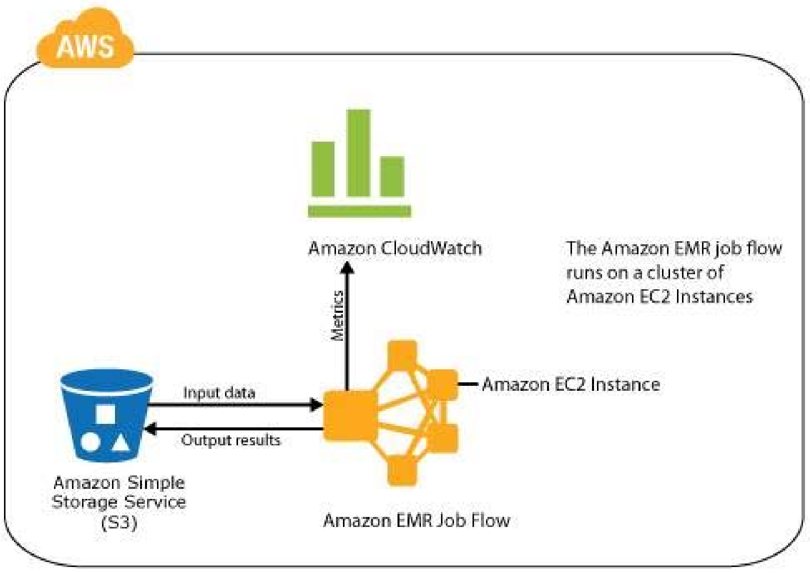

AWS
Amazon CloudWatch
The Amazon EMR job flow runs on a cluster of Amazon EC2 Instances
Metrics
Amazon EC2 Instance
Input data
Output results
Amazon Simple Storage Service (S3)
Amazon EMR Job Flow

Figure 12. Amazon EMR Interaction with Other AWS Services. Source: [33].

The first test was to validate performance of the MapReduce job on a single WARC file. Unlike before, when this file was downloaded to local storage, this test involved specifying an input path pointing to the Common Crawl public data set. The following is an example input path for a single WARC file:

```
s3n://aws-publicdatasets/common-crawl/crawl-data/CC-MAIN-
2016-07/segments/1454701145519.33/warc/CC-MAIN-
20160205193905-00000-ip-10-236-182-209.ec2.internal.warc.gz
```

This path points to a single file from a 2016 crawl. In order to analyze this file, this specified path becomes an input to the custom JAR arguments as seen in Appendix I. Some of the key job characteristics and performance data are listed in Table 3.

Table 3. Single WARC File Job Performance Characteristics

Category	Characteristic	Value
Hardware Instances:	Master	m3.xlarge (1)
	Core	i2.2xlarge (3)
Processing Time:	Elapsed Time	20 minutes
	Normalized Instance Hours	56 hours
File Size:	Input	887.5 MB (compressed)
	Output	1.99 GB (uncompressed)

The data in Table 3 is broken down into three categories of information. The first category, hardware instances, relates to the EC2 instance created to perform the MapReduce job. The master instance is the instance which managers the cluster and includes running the *namenode* service for the MapReduce job. The core instance contains all the nodes of the cluster; this includes running of the *datanodes* and is where the map and reduce tasks are executed [33]. The second category, processing time, has to do with time metrics related to job completion. Elapsed time refers to the real time that the MapReduce job took to complete, whereas the normalized instance hours relates to the computational complexity of the job. More specifically, this is related to type of size of the instances selected for the cluster [33]. As the cluster size increases, more parallelization is occurring and, therefore, more processing occurs in the same period of elapsed time. The last category, file size, corresponds to the input and output file sizes. The first two categories provide a point of comparison between this and any subsequent jobs. The last category shows that this job resulted in the same output size as the single WARC file job performed on the single node (local). The following is a small excerpt from the output:

```
montereyherald|http://www.montereyherald.com/20121113/spca-
for-monterey-county-urges-adoption-of-companion-
horses|405,1206,|

montereysuggest|http://monterey.org/en-us/City-
Hall/Newsroom/ItemId/166|671,|

monterey|http://digital.library.okstate.edu/chronicles/v019
/v019p014.html|9810,9627,9541,8599,8749,8891,8975,9346,|htt
p://www.globusjourneys.com/Vacation-Packages/Tour-North-
America/National-
Parks/?year=2012&gaparm=/Product.aspx?trip=3AVF,.,/Product.
aspx?trip=3AD|3717,1609,3709,1405,1603,1473,1482,|http://ia
n.umces.edu/press/location/rookery_bay/|671,|http://caselaw
.findlaw.com/us-supreme-
court/165/675.html|7302,|http://www.reforma.org/elections_s
p13|337,|http://www.thecarconnection.com/make/2003,jaguar,p
rice-range-0-55000|1141,|
```

Based on this output, we see that the MapReduce job successfully performed as expected on AWS. The next step in testing was to attempt to process multiple WARC files.

In order to test processing multiple WARC files, a full segment of crawl files was selected. In this case, the segment that was chosen was comprised of one hundred WARC files. The resulting output file sizes for this analysis are shown in Table 4. This data is the result of processing a segment with a combined input file size of approximately 64.98 GB. The size of the output files comes to a total of 140.13 GB, with the largest individual output file at 14.1 GB. As previously mentioned, one significant factor in this difference between input and output file size is that the output files are not compressed, while the input files are compressed. Some of the key job characteristics and performance data are listed in Table 5.

This job was created using a larger number of instances of a higher performance level. The single segment WARC job was completed in 336 normalized instance hours as compared to the previous job of a single WARC file in 56 hours. Additionally, the output file size (140.13 GB) is significantly greater for this job as compared to the output file size of 1.99 GB from the previous job. Now that the MapReduce jobs have been verified to function within AWS, it was worth investigating compression of the output.

Table 4. Inverted Index Output File Size from Processing a Crawl Segment

File	Character	Size		File	Character	Size
00000	0	255 MB		00019	j	1.9 GB
00001	1	429.5 MB		00020	k	1.7 GB
00002	2	589.3 MB		00021	l	5.5 GB
00003	3	438 MB		00022	m	8 GB
00004	4	298.9 MB		00023	n	4 GB
00005	5	286.5 MB		00024	o	4.1 GB
00006	6	212.7 MB		00025	p	10 GB
00007	7	296.7 MB		00026	q	798.9 MB
00008	8	222.1 MB		00027	r	6.6 GB
00009	9	208.8 MB		00028	s	14.1 GB
00010	a	8.8 GB		00029	t	7.8 GB
00011	b	6.5 GB		00030	u	2.9 GB
00012	c	13.2 GB		00031	v	2.6 GB
00013	d	6.4 GB		00032	w	4.3 GB
00014	e	5.3 GB		00033	x	422.7 MB
00015	f	5.8 GB		00034	y	1.4 GB
00016	g	4.5 GB		00035	z	566.4 MB
00017	h	5.4 GB		00036	(other)	0 MB
00018	i	4.3 GB				

Table 5. Single Segment Job Performance Characteristics

Category	**Characteristic**	**Value**
Hardware Instances:	Master	m3.2xlarge (1)
	Core	i2.8xlarge (5)
Processing Time:	Elapsed Time	43 minutes
	Normalized Instance Hours	336 hours
File Size:	Input	64.98 GB (uncompressed)
	Output	140.13 GB (uncompressed)

In order to alter the compression setting of a MapReduce job, the job configuration must be changed. There are a number of configuration variables that are applicable to compression [10]; however, in this case the goal was to only compress the final output from the *Reducers*. In order to do this, two job configuration variables required manipulation. The first variable was *mapreduce.output.fileoutputformat.compress*, which has a default value of *false* indicating no compression [10]. By updating the job configuration setting and setting this variable equal to *true*, we tell the system to compress the output of the reducers. The second variable that needed to be updated was *mapreduce.output.fileoutputformat.compress.codec*. This variable indicates which type of compression codec to use; for this thesis the GZIP compression codec was utilized [10]. After modifying these job configuration settings, we created and ran a new JAR file on the same single segment of WARC files as before. With this change, the resulting output was reduced from 140.13 GB to 59.02 GB, with the largest individual output file being 6 GB.

At this point, it was clear that the MapReduce job worked as designed and was able to create the inverted index relationship across multiple crawl files. Before attempting to process an even greater portion of the crawl data, we evaluated ten WARC files in order to try to establish a relationship between input file space and output file space. The key job characteristics related to this evaluation are given in Table 6.

Table 6. Ten WARC Files Job Performance Characteristics

Category	**Characteristic**	**Value**
Hardware Instances:	Master	m3.2xlarge (1)
	Core	i2.8xlarge (5)
Processing Time:	Elapsed Time	21 minutes
	Normalized Instance Hours	336 hours
File Size:	Input	8.1363 GB (uncompressed)
	Output	7.4620 GB (compressed)

Additionally, the test for one WARC file was reperformed with the MapReduce job performing compression on the output. The overall results of the combination of these three tests are shown in Table 7 and plotted in Figure 13:

Table 7. Comparison of Input and Output File Size for Three Tests

No. of Files	**Input Size**	**Output Size**
1	0.8875 GB	0.7464 GB
10	8.1363 GB	7.4520 GB
100	64.9778 GB	59.2027 GB

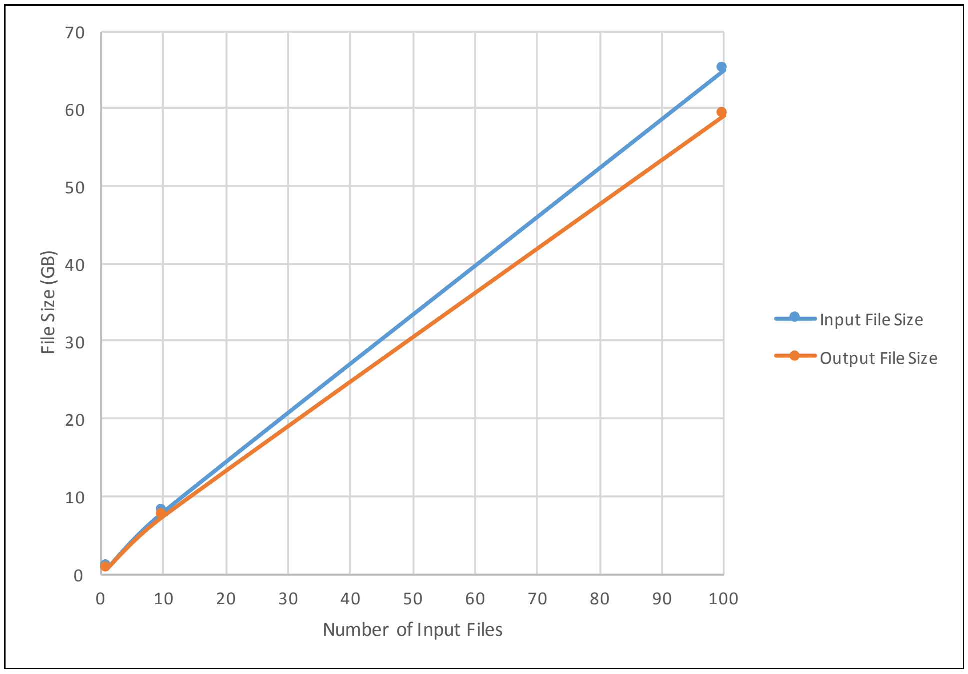


Figure 13. Relationship Between Number of Input Files and Associated Input/Output File Size

From these results, it appears as though there is a nearly linear relationship between the number of input files and the resulting output file size. Initially, it was thought that there may be a less-than-linear relationship as the input space increased due to the reduced number of new words encountered. This assumption may correlate to the slight bend between one and ten files; however, upon further investigation, the URL content drives the output file size of the inverted index relationship. Taking a small portion of previous output provided as an example:

```
monterey|http://digital.library.okstate.edu/chronicles/v019
/v019p014.html|9810,9627,9541,8599,8749,8891,8975,9346,|
```

By performing a cursory visual inspection of the output, we see that it appears that the majority of the terms are often less than ten characters in length, whereas the URL is often greater than three times this amount. Because each record entry contained within a WARC file is from a unique URL, as each term is found, the inverted index entry grows at least by the length of the URL. It then became clearer as to why the tests indicate a nearly linear relationship between the number of input files and the output file size.

This relationship in output file size growth was not anticipated, and the MapReduce job was not optimally designed and set up with respect to the number of *map* and *reduce* tasks. In addition, the EC2 cluster was not optimized based on the input space. Consequently, it was unclear as to how large this job could scale to before failure occurred due to the exhaustion of computing resources. Based on the size of the output files for one crawl segment and the fact that a single crawl may contain a hundred individual segments, it was expected that the use of 37 *Reducers* would eventually lead to job failure. This was later verified by attempting to run the same MapReduce job across three segments.

When attempting to run this job across multiple segments, totaling approximately 1 TB of input data, the job failed to complete. Specifically, the *Reducers* fail when they experience a *Java Heap Space Error*. Essentially, this means that the Java Virtual Machine (JVM) running the Java application has ran out of heap memory [34]. In other words, the JVM requires more memory to run than has been allocated to its heap space. Java heap space is a modifiable attribute but is not a part of the job configuration of a MapReduce job [35]. The default heap space is a configuration setting associated with the host machine. In this case, it depends on the instance type used when setting up the cluster on AWS and is modifiable by additional configuration steps [33]. While the methods to optimize the MapReduce job and computing resources were not investigated as a part of this thesis, multiple basic modifications to the job were attempted in order to evaluate courses of action.

In efforts to overcome the heap space error, two variables were manipulated. The first variable was that of the EC2 instance type [36]. Specifically, different models of the R3 Memory Optimized (8xlarge) and I2 Storage Optimized (8xlarge) instances were attempted. When configuring each of these instances, both were evaluated utilizing five and ten core instances. In addition to trying different instance types, the second variable considered was the number of *Reducers* exercised. The Apache Hadoop Application Program Interface (API) documentation provides a general rule of thumb for the *setNumReduceTasks()* method [10]. The maximum number of reduce tasks evaluated as a part of this testing was two hundred. None of these efforts changed the overall outcome of the MapReduce job. Each time the memory error was received, it occurred after the completion of the map phase and during the reduce phase of the job; however, depending on the cluster makeup and number of *Reducers*, the point at which the failure occurred varied within the reduce phase. This was verified by reviewing the output logs from the jobs.

The only variable not evaluated was that of changing the heap space configuration settings. Based on AWS documentation, it appears as though this can be accomplished by creating a JSON file containing the appropriate configuration commands [33]. This file is then subsequently uploaded when configuring the cluster settings. It is possible that increasing the Java heap space will allow indexing of a larger set of WARC files; however, it is also possible that attempting to index an entire crawl with one MapReduce job requires more resources than available within the EC2 instances even if optimized for the job. This then requires taking a different approach in order to create an entire inverted index from a complete crawl.

One possible course of action is to perform the MapReduce job as is but on smaller subsets of the crawl files. Another possible course of action is to approach the index in a slightly different manner. For example, create a MapReduce job to index all the URLs from the input space and use the resulting output as an input to a second MapReduce job, which performs the inverted index on the same input space. By following this sequence of MapReduce jobs, we accomplish a level of URL compression prior to processing the final output, thus reducing the overall output size. This sequence is a method of cascading MapReduce jobs in order to achieve the desired end state. An example of how to index URLs using a global counter for the *Reducers* is provided in the Supplemental Material (Appendix K).

C. SUMMARY

The results obtained from evaluation of the WARC inverted index MapReduce job was presented in this chapter. This included evaluation of performance on a single machine, as well as performance on AWS. In this analysis, the indexing tool was found to correctly process varying quantities of WARC files from different file structures and ultimately produced the desired inverted index relationship; however, once the job was scaled to larger portions of a crawl set, the MapReduce job failed due to errors associated with the Java heap space. Finally, a discussion was presented of some of the efforts associated with addressing this error, as well as potential courses of action for correction.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

The ultimate goal of the research project this thesis supports is to create a very robust search capability for the Common Crawl data set. The first step in this process, and the primary goal of this thesis, was to develop the ability to create an inverted index from the Common Crawl dataset. A MapReduce job was implemented that successfully traverses WARC files, extracts the payload of the appropriate record entries, and creates an enhanced inverted index relationship consisting of the term, the URL located on, and the relative location within, the respective URL. This capability was demonstrated on individual WARC files, as well as a crawl segment consisting of one hundred WARC files; however, further work is necessary to scale this capability to an entire crawl.

A. SUMMARY

A building block approach in the development of a tool capable of creating an enhanced inverted index relationship from Common Crawl WARC files was employed. This process began with the exploration of the Common Crawl file set, the analysis of data within the Hadoop Distributed File System, and resulted in the development of a series of MapReduce jobs with increasing capability. The first of these tools consisted of simple MapReduce jobs that performed a word count as well as creating an inverted index from text files. The second set of tools transitioned to conducting the same set of tasks but on WET files. Finally, this process concluded with the development of the MapReduce jobs capable of conducting a word count and creating an inverted index relationship on WARC files.

B. SIGNIFICANT RESULTS

The MapReduce job developed here to create an enhanced inverted index relationship can be successfully utilized to process WARC files. When provided the appropriate input path, this job can correctly process multiple WARC files and is capable of identifying crawl response records with an HTML payload. The body of this payload is then evaluated, with the resulting job output containing key-value pairs in the form of <term, (URL, term relative position)>. These key-value pairs are ultimately formed into the final inverted index relationship where each term is listed with every URL and its relative location on the associated URL.

During the course of evaluation it was determined that the MapReduce job, while able to process numerous WARC files, is not able to process an entire crawl file set at once. When attempting to run the job at this scale, a Java heap space error is received during the reduce phase. Ultimately, this means that some portion of the Java applications running the *Reducers* is exceeding the allotted heap space memory. This results in two conclusions: (1) the configuration of the job, as well as the cluster instances running the job, are not optimally configured to handle the job as is, or (2) the process of creating an inverted index from an entire crawl file set is too large to be accomplished by one MapReduce job.

The first conclusion is certainly true, since determining these settings was not the goal of this research; however, through the course of evaluation, it appears that the most likely configuration settings to evaluate for improvement are the number of reduce tasks chosen in the MapReduce job, the number and type of instances selected for the EC2 cluster, and the Java heap space configuration of the instances in the EC2 cluster. Even if this action alone does not create the environment capable of processing an entire crawl file set with one job, it will ultimately improve the performance of any future MapReduce jobs.

If the second conclusion is true, then there are two initial courses of action to pursue. As previously discussed, index compression is an area for improvement and will ultimately make a search capability more efficient. One approach would be to first process URLs of the input space and implement a compression method, such as serialization. The results from this MapReduce job can then become an input to a subsequent MapReduce job spanning the same input space. The serialization can then be used as a replacement of the entire URL within the inverted index, reducing the output file size. Cascading MapReduce jobs in this fashion may allow for an indexing MapReduce job to span an entire crawl file set. A second approach is to break up the input file space into multiple MapReduce jobs and then work to combine their output. This approach is a viable option to expand the input space in the event that improvements in job configuration and index compression still do not reduce the application memory to within the system constraints.

C. RECOMMENDATIONS FOR FUTURE WORK

This thesis research represents the first step in a much larger research project. There are numerous opportunities for future work, both in regards to the specific focus of this thesis as well as in respect to the greater research project.

The first area of continued work is in the inverted index for the Common Crawl dataset. Topics in this area are related to making improvements to the performance of the MapReduce job and inverted index so that an inverted index can be generated from an entire crawl set. Additionally, this area includes exploring techniques to improve the robustness of the index by identifying and including other meta-data of interest. The following three topics are specifically identified:

1. Determining how to optimize the MapReduce job with respect to maps tasks, reduce tasks, and EMR hardware configuration, (i.e., EC2 instance settings). This should also include any limitations on the input size to determine whether or not an entire crawl can be processed by a single MapReduce job or if it requires multiple jobs.

2. Determining methods to reduce the resulting inverted index file size. Specifically, exploring index compression techniques. A starting point would be a serialization of the URLs contained within the input files and, subsequently, using that serialization in the inverted index relationship rather than the complete URL.

3. Identifying and incorporating other meta-data of interest to include in the inverted index relationship. This could include other HTML meta tags or exploring the use of RDFa, Microdata, or other Microformats. The ability to identify and extract these elements of a website could lead to further enhancements to the inverted index relationship. This would, subsequently, result in a more robust search capability.

The second area of continued work is the development of a search/query tool. The WARC inverted index created from this thesis is in a searchable form; however, it does not include the additional elements needed to create a search engine. Specifically, this includes a query tool, a method to evaluate and rank the results, and a results interface.

A final area of continued work is in the area of improved WARC payload evaluation. This not only includes finding ways to improve analysis of HTML but also finding ways to evaluate other types of response records that may not be in HTML. Additionally, there are opportunities to improve pattern matching and filtering techniques with respect to the payload content. This would also help to reduce the size of the output files as nonsensical terms are removed.

APPENDIX A. BASIC WORD COUNT

```
package edu.nps.thesisProject.projectfiles;

/**
 * File:    basicWordCount.java
 * Purpose: This file demonstrates a basic implementation of a
 * MapReduce job.  The function performed is that of a word count.
 * The expected input is in the form of a files containing plain text.
 *
 * Input:   text file(s)
 * Output:  <word1> <count>
 *          <word2> <count>
 *
 * @author adcoudra1@nps.edu
 */

// Java Packages
import java.io.IOException;
import java.util.*;

// Hadoop Packages
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class basicWordCount {

  public static class basicWordCountMap extends
                           Mapper<Object, Text, Text, IntWritable> {

    // create commonWords, a list of strings for common words to
    //    exclude from the counting process
    // asList() creates a fixed size list initialized as specified
    List<String> commonWords = Arrays.asList("the", "a", "an", "and",
                  "of", "to", "in", "am", "is", "are", "at", "not");

    @Override
    public void map(Object key, Text value, Context context)
                        throws IOException, InterruptedException {
    /*
     * map() is the method that performs a tokenizer job, processing
     *    one line at a time and splitting the line into tokens based
     *    on identified characters
     */

      // create line, a String from the text of the map value
      // toString() converts the text (value) back to string
      String line = value.toString();

      // create a string tokenizer for the specified String line
      // StringTokenizer(String str, String delim) allows the
      //    application to break a string into tokens based on
      //    specified delimiters
      StringTokenizer tokenizer = new StringTokenizer(line,
                                          " \t,;.?!-:@[](){}_*/");

      // iterate through all words (tokens) in the line
      // take the next token and form key value pair
      // hasMoreTokens() tests for more tokens in the tokenizer's
      //    string and returns a boolean
      while (tokenizer.hasMoreTokens()) {

        // create nextToken, a String for the next token
        // nextToken() returns the next token from the tokenizer's
        //    string
        String nextToken = tokenizer.nextToken();

        // test to see if token is contained in commonWords, if so do
        //    not create key value pair
        // toLowerCase() converts all characters in string to lower
        //    case
        // trim() returns a copy of the string with all leading and
        //    trailing whitespace removed
        // contains() tests string for specified character sequence and
        //    returns a boolean
        if (!commonWords.contains(nextToken.trim().toLowerCase())) {

          // send key value pair to output collector to pass to the
          //    reducer
          // key value pair is <word, 1>
          // write() generates an output key value pair
          context.write(new Text(nextToken), new IntWritable(1));
        } // end of if (!commonWords…)
      } // end of while (tokenizer.hasMoreTokens()…)
    } // end of map ()
  } // end of basicWordCountMap

  public static class basicWordCountReduce extends
                        Reducer<Text, IntWritable, Text, IntWritable> {

    @Override
    public void reduce(Text key, Iterable<IntWritable> values,
            Context context) throws IOException, InterruptedException {
    /*
     * reduce is the method that accepts key value pairs from the
     *    mappers, sums up the values for each key and produces a final
     *    output
     */

      // create sum, an integer to track the count for each key,
      //    initialized to zero
      int sum = 0;

      // create an IntWritable val that iterates through IntWritable
      //    values
      // this loop iterates through each value passed for a given key
      for (IntWritable val : values) {

        // add each value for a given key to get the total sum
        // get() returns the integer value of the IntWritable
        sum += val.get();
      } // end for ()

      // send the final key value pair to output collector
      // key value pair is <key, sum>
      // write() generates an output key value pair
      context.write(key, new IntWritable(sum));
    } // end reduce ()
  } // end basicWordCountReduce

  public static void main(String[] args) throws Exception {
  /*
   * main is the overall driver method that triggers the mapreduce job
   *    in hadoop
   */

    // create a job and assign a name (basicwordcount) for
    //    identification
    // getInstance(Configuration conf, String jobName) creates a job
    //    with the specified configuration and job name
    // Configuration() provides access to configuration parameters
    Job job = Job.getInstance(new Configuration(), "basicwordcount");

    // configure the key and value output class for the job
    // setOutputKeyClass() sets the key class for the job output data
    // setOutputValueClass() sets the value class for the job output
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    // configure the mapper, combiner (optional) and reducer classes
    // setMapperClass() sets the mapper for the job
    // setCombinerClass() sets the combiner class for the job
    // setReducerClass() sets the reducer class for the job
    // provide mapper, combiner, and reducer class names
    job.setMapperClass(basicWordCountMap.class);
    job.setCombinerClass(basicWordCountReduce.class);
    job.setReducerClass(basicWordCountReduce.class);

    // configure the hdfs input and output directory as fetched from
    //    the command line
    // Path() names a file or directory based on the argument string
    // setInputPaths() sets the array of Path as the list of inputs for
    //    the job
    // setOutputPath() sets the array of Path as the output directory
    //    for the job
    //
```

```
    // set the input and output path for the job
    // To take specified file paths from user, use:
    //    String inputPath = args[0];
    //    String outputPath = args[1];
    //
    // To take predefined file paths, use:
    //    String inputPath = "data/text/*.*";
    //    String outputPath = "output/";
    String inputPath = args[0];
    String outputPath = args[1];

    FileInputFormat.setInputPaths(job, new Path(inputPath));
    FileOutputFormat.setOutputPath(job, new Path(outputPath));

    // configure the jar class
    // setJarByClass() sets the jar by finding where a given class came
    //    from
    job.setJarByClass(basicWordCount.class);

    // submit the job to mapreduce and wait for completion
    // System is a class that provides standard input, output and error
    //    streams
    // exit() terminates the currently running java virtual machine
    // waitForCompletion() submits the job to the cluster and waits for
    //    it to finish
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  } // end main ()
} // end basicWordCount
```

APPENDIX B. BASIC INVERTED INDEX

```java
package edu.nps.thesisProject.projectfiles;

/**
 * File:    basicInvertedIndexCount.java
 * Purpose: This file demonstrates a basic implementation of a
 *   MapReduce job.  The function performed is that of generating an
 *   inverted index, to include the additional information of relative
 *   word count.  The expected input is in the form of a files
 *   containing plain text.
 *
 * Input:   text files
 * Output:  <word1>@<filename1> <count>
 *          <word1>@<filename2> <count>
 *          <word2>@<filename1> <count>
 *
 * @author adcoudra1@nps.edu
 */

// Java Packages
import java.io.IOException;
import java.util.*;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

// Hadoop Packages
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class basicInvertedIndexCount {

  public static class basicInvertedIndexCountMap extends
                        Mapper<LongWritable, Text, Text, IntWritable> {

    // create commonWords, a list of strings for common words to
    //    exclude from counting process
    // asList() creates a fixed size list initialized as specified
    private static List<String> commonWords = Arrays.asList("the", "a",
                        "an", "and", "of", "to", "in", "am", "is",
                        "are", "at", "not");

    // create pattern, a compiled representation of a regular
    //    expression
    // Pattern is the overarching class
    // compile() creates the pattern based on the provided flags
    // \w indicates a word character, short for [a-zA-Z_0-9]
    // \\w the first \ is the escape character so that Java will
    //    recognize '\w' as the input
    // + indicates occurring one or more times, short for {1,}
    private static Pattern pattern = Pattern.compile("\\w+");

    // create word, the eventual map output key
    private Text word = new Text();

    // create docID, to be used to contain the current file name
    private String docID = new String();

    // create count, the output value equal to 1
    private IntWritable count = new IntWritable(1);

    public void map(LongWritable key, Text value, Context context)
                        throws IOException, InterruptedException {
    /*
     * map is the method that performs a tokenizer job, processing one
     *    line at a time and splitting the line into tokens based on
     *    identified characters
     */

      // create matcher to perform regex operation on the input String
      // matcher() matches the input sequence against the provided
      //    pattern
      // toString() converts the text (value) into a String
      Matcher matcher = pattern.matcher(value.toString());

      // create valueBuilder a mutable sequence of characters
      // StringBuilder() constructs a string builder with no characters
      //    in it
      StringBuilder valueBuilder = new StringBuilder();

      // obtain the file name of the current file being processed
      // getInputSplit() gets the input split for this map
      // getPath() gets the file containing this split's data
      // getName() returns the final component (filename) of this file
      //    path
      docID = ((FileSplit)context.getInputSplit()).getPath().getName();

      // iterate through all matches of the regex pattern found in the
      //    input take the next match, convert to lower case and check
      //    against common words, if not contained in common words,
      //    form and emit the key value pair
      // find() attempts to find the next subsequence of the input
      //    sequence that matches
      while(matcher.find()) {

        // create matchedKey from each matched pattern converted to
        //    lower case
        // group() returns the input subsequence matched by the
        //    previous match
```

```
        // toLowerCase() converts the input to all lower case
        String matchedKey = matcher.group().toLowerCase();

        // test to see if the key is contained in commonWords, if so do
        //    not create key value pair
        // trim returns a copy of the string with all leading and
        //    trailing whitespace removed
        // contains() tests string for specified character sequence and
        //    returns a boolean
        if (!commonWords.contains(matchedKey)) {

          // add the matched key to valueBuilder (this will be first
          //    element)
          // append() appends the string representation of the argument
          valueBuilder.append(matchedKey);

          // append @ to valueBuilder
          valueBuilder.append("@");

          // append the current filename to valueBuilder
          valueBuilder.append(docID);

          // set the output key (word) to the contents of valueBuilder
          // set() assigns the input String to the contents
          this.word.set(valueBuilder.toString());

          // send key value pair to output collector to pass to the
          //    reducer
          // key value pair is <word@filename, 1>
          // write() generates an output key value pair
          context.write(this.word, this.count);

          // clear the contents of valueBuilder
          // setLength() sets the length of the character sequence
          valueBuilder.setLength(0);
        } // end if (!commonWords…)
      } // end while (matcher.find())
    } // end map ()
  } // end basicInvertedIndexCountMap

  public static class basicInvertedIndexCountReduce extends
                        Reducer<Text, IntWritable, Text, IntWritable> {

    // create an IntWritable to track the count sum for each key
    private IntWritable wordSum = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
            Context context) throws IOException, InterruptedException {
    /*
     * reduce is the method that accepts key value pairs from the
     *    mappers, sums up the values for each key and produces a final
     *    output
     */

      // create an integer to track the count sum for each key,
    //    initialized to zero
    int sum = 0;

    // create an IntWritable val that iterates through IntWritable
    //    values
    // this loop iterates through each value passed for a given key
    for (IntWritable val : values) {

      // add each value for a given key to get the total sum
      // get() returns the integer value of the IntWritable
      sum += val.get();
    } // end for ()

    // set the value of wordSum to the contents of sum
    this.wordSum.set(sum);

    // send the final key value pair to output collector
    // key value pair is <key, sum>
    // write() generates an output key value pair
    context.write(key, this.wordSum);
  } // end reduce ()
} // end basicInvertedIndexCountReduce

public static void main(String[] args) throws Exception {
/*
 * main is the overall driver method that triggers the mapreduce job
 *    in hadoop
 */

  // create a job and assign a name (basicinvertedindexcount) for
  //    identification
  // getInstance(Configuration conf, String jobName) creates a job
  //    with the specified configuration and job name
  // Configuration() provides access to configuration parameters
  Job job = Job.getInstance(new Configuration(),
                                      "basicinvertedindexcount");

  // configure the key and value output class for the job
  // setOutputKeyClass() sets the key class for the job output data
  // setOutputValueClass() sets the value class for the job output
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  // configure the mapper, combiner (optional) and reducer classes
  // setMapperClass() sets the mapper for the job
  // setCombinerClass() sets the combiner class for the job
  // setReducerClass() sets the reducer class for the job
  // provide mapper, combiner, and reducer class names
  job.setMapperClass(basicInvertedIndexCountMap.class);
  job.setCombinerClass(basicInvertedIndexCountReduce.class);
  job.setReducerClass(basicInvertedIndexCountReduce.class);

  // configure the hdfs input and output directory as fetched from
  //    the command line
  // Path() names a file or directory based on the argument string
```

```
    // setInputPaths() sets the array of Path as the list of inputs for
    //    the job
    // setOutputPath() sets the array of Path as the output directory
    //    for the job
    //
    // set the input and output path for the job
    // To take specified file paths from user, use:
    //    String inputPath = args[0];
    //    String outputPath = args[1];
    //
    // To take predefined file paths, use:
    //    String inputPath = "data/text/*.*";
    //    String outputPath = "output/";
    String inputPath = args[0];
    String outputPath = args[1];

    FileInputFormat.setInputPaths(job, new Path(inputPath));
    FileOutputFormat.setOutputPath(job, new Path(outputPath));

    // configure the jar class
    // setJarByClass() sets the jar by finding where a given class came
    //    from
    job.setJarByClass(basicInvertedIndexCount.class);

    // submit the job to mapreduce and wait for completion
    // System is a class that provides standard input, output and error
    //    streams
    // exit() terminates the currently running java virtual machine
    // waitForCompletion() submits the job to the cluster and waits for
    //    it to finish
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  } // end main ()
} // end basicInvertedIndexCount
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. WET FILE WORD COUNT

```
package edu.nps.thesisProject.projectfiles;

/**
 * File:    WETWordCount.java
 * Purpose: This file provides for a tool runner implementation of a
 *   MapReduce job.  The function performed is that of conducting a
 *   word count.  The expected input is in the form of WARC record
 *   entries from a WET file.
 *
 * Driver for: WETWordCountMap.java
 *
 * Input:   WET files
 * Output:  <word> <count>
 *
 * @author adcoudra1@nps.edu
 */

// WARC Utilities
import edu.nps.thesisProject.warcutils.*;

// Hadoop Packages
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.lib.reduce.LongSumReducer;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

// Log4j Package
import org.apache.log4j.Logger;

public class WETWordCount extends Configured implements Tool {

  // create a Logger object called LOG that retrieves a logger named
  //    after the class
  private static final Logger LOG =
                              Logger.getLogger(WETWordCount.class);

  public static void main(String[] args) throws Exception {
  /*
   * main is the overall driver method that triggers the mapreduce job
   *    in hadoop
   */

    // RUN(CONFIGURATION conf, TOOL tool, STRING[] args)
    // runs the given Tool by Tool.run(String[]) after parsing given
    //    generic arguments
    // RUN(STRING[] args)
    // executes the command given the arguments
    //
    // overall this is equivalent to creating a new job and assigning
    //    the input and output path
    //
    // returns the exit code of the Tool.run(String[]) method
    int res = ToolRunner.run(new Configuration(),
                                       new WETWordCount(), args);

    // terminates the currently running java virtual machine
    System.exit(res);
  } // end main ()

  @Override
  public int run(String[] args) throws Exception {
  /*
   * Builds and runs the Hadoop job.
   * return 0 if the Hadoop job completes successfully and 1 otherwise.
   */

    // set the input and output path for the job
    // To take specified file paths from user, use:
    //    String inputPath = args[0];
    //    String outputPath = args[1];
    //
    // To take predefined file paths, use:
    //    String inputPath = "data/wet/*.warc.wet.gz";
    //    String outputPath = "output/";
    //
    // Example paths for AWS:
    //    String inputPath = "s3n://aws-publicdatasets/common-
    //    crawl/crawl-data/CC-MAIN-2013-48/segments/
    //    1386163035819/wet/CC-MAIN-20131204131715-00000-ip-10-33-
    //    133-15.ec2.internal.warc.wet.gz";
    //
    //    String inputPath = "s3n://aws-publicdatasets/common-
    //    crawl/crawl-data/CC-MAIN-2013-48/segments/
    //    1386163035819/wet/*.warc.wet.gz";
    String inputPath = args[0];
    String outputPath = args[1];

    // Configuration processed by ToolRunner
    // getConf() returns the configuration used by this object
    Configuration conf = this.getConf();

    // create a job and assign a name (wetwordcount) for identification
    // getInstance(Configuration conf, String jobName) creates a job
    //    with the specified configuration and job name
    Job job = Job.getInstance(conf, "wetwordcount");

    // configure the InputFormat for the job as WarcInputFormat
    // setInputFormatClass() sets the InputFormat
    job.setInputFormatClass(WarcInputFormat.class);

    // configure the OutputFormat for the job as TextOutputFormat
    // setOutputFormatClass() sets the OutputFormat
    job.setOutputFormatClass(TextOutputFormat.class);

    // configure the key and value output class for the job
    // setOutputKeyClass() sets the key class for the job output data
    // setOutputValueClass() sets the value class for the job output
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(LongWritable.class);

    // configure the mapper, combiner (optional) and reducer classes
    // setMapperClass sets the mapper for the job
    // setCombinerClass sets the combiner class for the job
    // setReducerClass sets the reducer class for the job
    // provide mapper, combiner, and reducer class names
    job.setMapperClass(WETWordCountMap.WETWordCountMapper.class);
    job.setCombinerClass(LongSumReducer.class);
    job.setReducerClass(LongSumReducer.class);

    // configure the hdfs input and output directory as fetched from
    //    the command line
    // Path names a file or directory based on the argument string
    // setInputPaths sets the array of Path as the list of inputs for
    //    the job
    // setOutputPath sets the array of Path as the output directory for
    //    the job
    FileInputFormat.addInputPath(job, new Path(inputPath));
    FileOutputFormat.setOutputPath(job, new Path(outputPath));

    // configure the jar class
    // setJarByClass() sets the jar by finding where a given class came
    //    from
    job.setJarByClass(WETWordCount.class);

    // log a message object with the info level (highlight progress of
    //    the application)
    LOG.info("Input path: " + inputPath);

    // submit the job to mapreduce and wait for completion, then return
    //    status
    // waitForCompletion() submits the job to the cluster and waits for
    //    it to finish
    return job.waitForCompletion(true) ? 0 : 1;
  } end run ()
} // end WETWordCount
```

```java
package edu.nps.thesisProject.projectfiles;

/**
 * File:    WETWordCountMap.java
 * Purpose: This file provides a MapReduce job that performs the
 *   function of a word count on WARC record entries from a WET file.
 *
 * Map for: WETWordCount.java
 *
 * Input:   WET files
 * Output:  <word1> <count>
 *          <word2> <count>
 *
 * @author adcoudra1@nps.edu
 */

// Java Packages
import java.io.IOException;
import java.util.StringTokenizer;

// Hadoop Packages
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

// Other Apache Packages
import org.apache.commons.io.IOUtils;
import org.apache.log4j.Logger;

// JWAT Packages
import org.jwat.common.Payload;
import org.jwat.warc.WarcRecord;

public class WETWordCountMap {

  // create a Logger object called LOG that retrieves a logger named
  //    after the class
  private static final Logger LOG =
                      Logger.getLogger(WETWordCountMapper.class);

  // enum is a special data type that enables for a variable to be a
  //    set of predefined constants
  protected static enum MAPPERCOUNTER {

    EMPTY_RECORDS,
    EXCEPTIONS,
    NON_PLAIN_TEXT,
    CURRENT_RECORD,
    NUM_TEXT_RECORDS
  } // end MAPPERCOUNTER

  protected static class WETWordCountMapper extends
          Mapper<LongWritable, WarcRecord, Text, LongWritable> {

    // create a string tokenizer for the specified string line
```

```
private StringTokenizer tokenizer;

// create outKey, the eventual map output key
private Text outKey = new Text();

// create outVal, the output value equal to 1
private LongWritable outVal = new LongWritable(1);

@Override
public void map(LongWritable key, WarcRecord value,
        Context context) throws IOException, InterruptedException {
/*
 * map is the method that performs a tokenizer job, processing
 *    one record at a time
 */

  // setStatus(String msg) sets the current status of the task to
  //    the given string
  context.setStatus(MAPPERCOUNTER.CURRENT_RECORD + ": " +
                                                       key.get());

  // try the following and if an error occurs, catch exception
  try {

    // only process text/plain content
    // value.header.contentTypeStr retrieves the "Content-Type:"
    //    field of the record header
    if ("text/plain".equals(value.header.contentTypeStr)) {

      // getCounter(Enum<?> counterName) gets the counter for the
      //    given counterName
      // increment(long incr) increments the counter by the given
      //    value
      context.getCounter(MAPPERCOUNTER.NUM_TEXT_RECORDS).
                                                increment(1);

      // Get the record payload
      // getPayload() returns the payload object for the record
      Payload payload = value.getPayload();

      // if payload is empty perform no operation
      // else process record for word count
      if (payload == null) {
        // NOP
      } // end if (payload == null)
      else {
        // create warcContent string from record payload
        // toString() converts the text (value) into a String
        // getInputStreamComplete() gets the InputStream to read
        //    the complete payload
        String warcContent =
              IOUtils.toString(payload.getInputStreamComplete());

        // if the record content is empty, increment counter
        // else process record content
```

```
            if (warcContent == null && "".equals(warcContent)) {
              context.getCounter(MAPPERCOUNTER.EMPTY_RECORDS).
                                                      increment(1);
            } // end if (warcContent is empty)
            else {
              // assign the warcConent to string tokenizer
              tokenizer = new StringTokenizer(warcContent);

              // iterate through all words (tokens) in the record, take
              //    next token and form key value pair
              // hasMoreTokens() tests for more tokens in the
              //    tokenizer’s string and returns a boolean
              while (tokenizer.hasMoreTokens()) {

                // set the outKey as the token
                outKey.set(tokenizer.nextToken());

                // send key value pair to output collector to pass to
                //    the reducer
                // key value pair is <Text outKey, 1>
                // write() generates an output key value pair
                context.write(outKey, outVal);
              } // end while (more tokens)
            } // end else (warcContent has content)
          } // end else (payload has content)
        } // end if (record is plaintext)
        else { // if the record content is not plaintext

          // when not plaintext, increment the counter
          context.getCounter(MAPPERCOUNTER.NON_PLAIN_TEXT).
                                                      increment(1);
        } // end else (record not plaintext)
      } // end try ()
      catch (Exception ex) {

        // if the try statement fails, catch the exception
        // error(String msg, Throwable t)
        // output the internal error statement
        LOG.error("Caught Exception", ex);

        // when exception occurs, increment the counter
        context.getCounter(MAPPERCOUNTER.EXCEPTIONS).increment(1);
      } // end catch
    } // end map ()
  } // end WETWordCountMapper
} // end WETWordCountMap
```

APPENDIX D. WET FILE INVERTED INDEX

```java
package edu.nps.thesisProject.projectfiles;

/**
 * File:    WETIndexLocationCustom.java
 * Purpose: This file provides for a tool runner implementation of a
 *   MapReduce job.  The function performed is that a creating an
 *   inverted index using a custom tuple data type.  The expected input
 *   is in the form of WARC record entries from a WET file.
 *
 * Driver for: WETIndexLocationCustomMap.java
 *
 * Input:   WET files
 * Output:  <word1> : <url1>@<location1>, <location2>,
 *                    <url2>@<location1>, <location2>,
 *          <word2> : <url1>@<location1>, <location2>,
 *                    <url2>@<location1>, <location2>,
 *
 * @author adcoudra1@nps.edu
 */

//WARC Utilities
import edu.nps.thesisProject.warcutils.*;

// Hadoop Packages
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

//Log4j Package
import org.apache.log4j.Logger;

public class WETIndexLocationCustom extends Configured implements
                                                              Tool {

  // create a Logger object called LOG that retrieves a logger named
  //    after the class
  private static final Logger LOG =
                      Logger.getLogger(WETIndexLocationCustom.class);

  public static void main(String[] args) throws Exception {
  /*
   * main is the overall driver method that triggers the mapreduce job
   *    in hadoop
   */
  // RUN(CONFIGURATION conf, TOOL tool, STRING[] args)
  // runs the given Tool by Tool.run(String[]) after parsing given
  //    generic arguments
  // RUN(STRING[] args)
  // executes the command given the arguments
  //
  // overall this is equivalent to creating a new job and assigning
  //    the input and output path
  //
  // returns the exit code of the Tool.run(String[]) method
  int res = ToolRunner.run(new Configuration(),
                          new WETIndexLocationCustom(), args);

  // terminates the currently running java virtual machine
  System.exit(res);
} // end main ()

@Override
public int run(String[] args) throws Exception {
/*
 * Builds and runs the Hadoop job.
 * return 0 if the Hadoop job completes successfully and 1 otherwise.
 */

  // set the input and output path for the job
  // To take specified file paths from user, use:
  //    String inputPath = args[0];
  //    String outputPath = args[1];
  //
  // To take predefined file paths, use:
  //    String inputPath = "data/wet/*.warc.wet.gz";
  //    String outputPath = "output/";
  //
  // Example paths for AWS:
  //    String inputPath = "s3n://aws-publicdatasets/common-
  //    crawl/crawl-data/CC-MAIN-2013-48/segments/
  //    1386163035819/wet/CC-MAIN-20131204131715-00000-ip-10-33-133-
  //    15.ec2.internal.warc.wet.gz";
  //
  //    String inputPath = "s3n://aws-publicdatasets/common-
  //    crawl/crawl-data/CC-MAIN-2013-48/segments/
  //    1386163035819/wet/*.warc.wet.gz";

  //    String inputPath = "s3n://aws-publicdatasets/common-
  //    crawl/crawl-data/CC-MAIN-2015-48/segments/
  //    1448398444047.40/wet/CC-MAIN-20151124205404-00000-ip-10-71-
  //    132-137.ec2.internal.warc.wet.gz";
  //    String outputPath = "s3://thesis-wet-custom-index/output";
  String inputPath = args[0];
  String outputPath = args[1];

  // Configuration processed by ToolRunner
  // getConf() returns the configuration used by this object
  Configuration conf = this.getConf();
```

```
// create a job and assign a name (wetindexlocationcustom) for
//    identification
// getInstance(Configuration conf, String jobName) creates a job
//    with the specified configuration and job name
Job job = Job.getInstance(conf, "wetindexlocationcustom");

// configure the InputFormat for the job as WarcInputFormat
// setInputFormatClass() sets the InputFormat
job.setInputFormatClass(WarcInputFormat.class);

// configure the Map OutputValue for the job as
//    termLocationCustomWritable
// setMapOutputValueClass() sets the MapOutputValue
job.setMapOutputValueClass(termLocationCustomWritable.class);

// configure the OutputFormat for the job as TextOutputFormat
// setOutputFormatClass() sets the OutputFormat
job.setOutputFormatClass(TextOutputFormat.class);

// configure the key and value output class for the job
// setOutputKeyClass() sets the key class for the job output data
// setOutputValueClass() sets the value class for the job output
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);

// configure the mapper, combiner (optional) and reducer classes
// setMapperClass sets the mapper for the job
// setCombinerClass sets the combiner class for the job
// setReducerClass sets the reducer class for the job
// provide mapper, combiner, and reducer class names
job.setMapperClass(WETIndexLocationCustomMap.
                          WETIndexLocationCustomMapper.class);
job.setReducerClass(WETIndexLocationCustomMap.
                          WETIndexLocationCustomReducer.class);

// configure the partitions of the key space
// setPartitionerClass() sets the partitions class for the job
// setNumReduceTasks() sets the number of reduce tasks for the job
// in this case, the number of reduce tasks is set to match the
//    overall partitioning
job.setPartitionerClass(WETIndexLocationCustomMap.
                          WETIndexLocationCustomPartitioner.class);
job.setNumReduceTasks(37);

// configure the hdfs input and output directory as fetched from
//    the command line
// Path names a file or directory based on the argument string
// setInputPaths sets the array of Path as the list of inputs for
//    the job
// setOutputPath sets the array of Path as the output directory for
//    the job
FileInputFormat.addInputPath(job, new Path(inputPath));
FileOutputFormat.setOutputPath(job, new Path(outputPath));
```

```
    // configure the jar class
    // setJarByClass() sets the jar by finding where a given class came
    //    from
    job.setJarByClass(WETIndexLocationCustom.class);

    // log a message object with the info level (highlight progress of
    //    the application)
    LOG.info("Input path: " + inputPath);

    // submit the job to mapreduce and wait for completion, then return
    //    status
    // waitForCompletion() submits the job to the cluster and waits for
    //    it to finish
    return job.waitForCompletion(true) ? 0 : 1;
  } // end run ()
} // end WETIndexCustomLocation
```

```
package edu.nps.thesisProject.projectfiles;

/**
 * File:    WETIndexLocationCustomMap.java
 * Purpose: This file provides a MapReduce job that performs the
 *   function of an inverted index on WARC record entries from a WET
 *   file, using a custom tuple data type.
 *
 * Map for: WETIndexLocationCustom.java
 *
 * Input:   WET files
 * Output:  <word1> : <url1>@<location1>, <location2>,
 *                    <url2>@<location1>, <location2>,
 *          <word2> : <url1>@<location1>, <location2>,
 *                    <url2>@<location1>, <location2>,
 *
 * @author adcoudra1@nps.edu
 */

// Java Packages
import java.io.IOException;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map.Entry;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

// Hadoop Packages
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;

// Other Apache Packages
import org.apache.commons.io.IOUtils;
import org.apache.log4j.Logger;

// JWAT Packages
import org.jwat.common.Payload;
import org.jwat.warc.WarcRecord;

public class WETIndexLocationCustomMap {

  // create a Logger object called LOG that retrieves a logger named
after the class
  private static final Logger LOG =
                Logger.getLogger(WETIndexLocationCustomMapper.class);

  // enum is a special data type that enables for a variable to be a
  //    set of predefined constants
  protected static enum MAPPERCOUNTER {
    EMPTY_RECORDS,
    EXCEPTIONS,
    NON_PLAIN_TEXT,
    CURRENT_RECORD,
    NUM_TEXT_RECORDS
  }

  protected static class WETIndexLocationCustomMapper extends
                            Mapper<LongWritable, WarcRecord, Text,
                                        termLocationCustomWritable> {

    // NOTE: Two different regex patterns are offered – keep one
    //    commented out
    // regex method (1):
    // create pattern, a compiled representation of a regular
    //    expression
    // Pattern is the overarching class
    // compile() creates the pattern based on the provided flags
    // \w indicates a word character, short for [a-zA-Z_0-9]
    // \\w the first \ is the escape character so that Java will
    //    recognize '\w' as the input
    // + indicates occurring one or more times, short for {1,}
//    private static Pattern pattern = Pattern.compile("\\w+");

    // regex method (2):
    // (?=[a-zA-Z0-9]+[a-zA-Z]) is a positive lookahead assertion
    //  ?=                      : positive lookahead
    //    [a-zA-Z0-9]+          : match any characters in the list, an
    //                            unlimited number of times
    //                [a-zA-Z]  : that contains at least one of the
    //                            specified characters
    //    => a single character in the specified list [] must be
    //       present
    // ([a-zA-Z0-9]+) is the first capturing group
    //               +          : between one and unlimited times, as
    //                            many times as possible
    //   a-z                    : a single character between lower case
    //                            a and z
    //      A-Z                 : a single character between upper case
    //                            A and Z
    //         0-9              : a single character between 0 and 9
    //    => this regex should match any combination of letters and
    //       numbers, excluding only number combinations
    private static Pattern pattern =
            Pattern.compile("(?=[a-zA-Z0-9]+[a-zA-Z])([a-zA-Z0-9]+)");

    private termLocationCustomWritable recordLocation =
                                    new termLocationCustomWritable();

    // create commonWords, a list of strings for common words to
    //    exclude from counting process
    // Arrays is a class containing methods for manipulating arrays
    // asList() creates a fixed size list initialized as specified
    List<String> commonWords = Arrays.asList("a", "about", "am", "an",
            "and", "are", "as", "at", "be", "by", "com", "de", "en",
            "for", "from", "how", "i", "in", "is", "it", "la", "not",
            "of", "on", "or", "that", "the", "this", "to", "was",
            "what", "when", "where", "who", "will", "with", "www");

    // create recordID, to be used to contain the current record ID
    private String recordID = new String();

    // create word, the eventual map output key
    private Text word = new Text();

    @Override
    public void map(LongWritable key, WarcRecord value,
            Context context) throws IOException, InterruptedException {
    /*
     * map is the method that performs a tokenizer job, processing one
     *    record at a time
     */

      // setStatus(String msg) sets the current status of the task to
      //    the given string
      context.setStatus(MAPPERCOUNTER.CURRENT_RECORD + ": " +
                                                            key.get());

      // try following and if an error occurs, catch exception
      try {

        // only process text/plain content
        // value.header.contentTypeStr retrieves the "Content-Type:"
        //    field of the record header
        if ("text/plain".equals(value.header.contentTypeStr)) {

          // getCounter(Enum<?> counterName) gets the Counter for the
          //    given counterName
          // increment(long incr) increments the counter by the given
          //    value
          context.getCounter(MAPPERCOUNTER.NUM_TEXT_RECORDS).
                                                      increment(1);

          // Get the record payload
          // getPayload() returns the payload object for the record
          Payload payload = value.getPayload();

          // if payload is empty perform no operation
          // else process record for word count
          if (payload == null) {
            // NOP
          } // end if (payload is null)
          else {

            // create warcContent string from record payload
            // toString() converts the text (value) into a String
            // getInputStreamComplete() gets the InputStream to read
            //    the complete payload
            String warcContent =
                  IOUtils.toString(payload.getInputStreamComplete());
```

```
// if the record content is empty, increment counter
// else process record content
if (warcContent == null && "".equals(warcContent)) {

  context.getCounter(MAPPERCOUNTER.EMPTY_RECORDS).
                                        increment(1);
} // end if (warcContent is empty)
else {

  // create matcher to perform regex operation on the input
  //    String
  // matcher() matches the input sequence against the
  //    provided pattern
  // toString() converts the text (value) into a String
  Matcher matcher = pattern.matcher(warcContent);

  // obtain the url of the current record being processed
  // value.header.warcTargetUriStr retrieves the
  //    "WARC-Target-URI:" field of the record header
  recordID = value.header.warcTargetUriStr;

  // initialize wordLocation to 0
  int wordLocation = 0;

  // iterate through all matches of the regex pattern found
  //    in the input take the next match, convert to lower
  //    case and check against common words, if not
  //    contained in common words, form and emit the key
  //    value pair find() attempts to find the next
  //    subsequence of the input sequence that matches
  while(matcher.find()) {

    // increment wordLocation
    wordLocation++;

    // create matchedKey from each matched pattern
    //    converted to lower case
    // group() returns the input subsequence matched by the
    //    previous match
    // toLowerCase() converts the input to all lower case
    String matchedKey =
              matcher.group().trim().toLowerCase();

    // test to see if token is contained in commonWords, if
    //    so do not create key value pair
    // toLowerCase() converts all characters in string to
    //    lower case
    // trim() returns a copy of the string with all leading
    //    and trailing whitespace removed
    // contains() tests string for specified character
    //    sequence and returns a boolean
    if (!commonWords.contains(matchedKey)) {

      // set the value of word to the matched word,
                //    concatenated with " : "
                word.set(new Text(matchedKey+" : "));

                // set the value of the custom datatype
                recordLocation.set(recordID, wordLocation);

                // send key value pair to output collector to pass to
                //    the reducer
                // key value pair is <word, (url, location)>
                // write() generates an output key value pair
                context.write(word, recordLocation);
              } // end if (not in commonWords)
            } // end while (matcher)

            // reset word counter
            wordLocation = 0;
          } // end else (warcContent has content)
        } // end else (payload not empty)
      } // end if (record is plaintext)
      else { // if the record content is not plaintext

        // when not plaintext, increment the counter
        context.getCounter(MAPPERCOUNTER.NON_PLAIN_TEXT).
                                                increment(1);
      } // end else record not plaintext
    } // end try ()
    catch (Exception ex) {

      // if the try statement fails, catch the exception
      // error(String msg, Throwable t)
      // output the internal error statement
      LOG.error("Caught Exception", ex);

      // when exception occurs, increment the counter
      context.getCounter(MAPPERCOUNTER.EXCEPTIONS).increment(1);
    } // end catch ()
  } // end map ()
} // end WETIndexCustomLocationMapper

public static class WETIndexLocationCustomReducer extends
          Reducer<Text, termLocationCustomWritable, Text, Text> {

  @Override
  public void reduce(Text key, Iterable<termLocationCustomWritable>
                                values, Context context) throws
                                IOException, InterruptedException {
  /*
   * reduce is the method that accepts key value pairs from the
   *    mappers, sums up the values for each key and produces a final
   *    output
   */

    // create map, a HashMap<K, V> of type <Text, Text>
    HashMap<Text, Text> map = new HashMap<Text, Text>();
```

```
// create tempList, a String that will be used to hold a
//    temporary hashmap value
String tempList = "";

// create val, of the custom type termLocationCustomWritable that
//    iterates through each termLocationCustomWritable value
// this loop iterates through each value passed for a given key
for (termLocationCustomWritable val : values) {

  // create recordID, and assign to it the RecordID (URL)
  //    associated with val
  // getRecordID() returns the Text recordID contained within the
  //    custom data type
  Text recordID = new Text(val.getRecordID());

  // create recordLoc, and assign to it the Location (word
  //    position) associated with val
  // getLocation() returns the IntWritable location contained
  //    within the custom data type
  // toString() converts the IntWritable to a String
  Text recordLoc = new Text(val.getLocation().toString());

  // check the map to see if it contains any values associated
  //    with the specified key
  // this checks to see to see whether or not a key value has
  //    been placed in the hashmap
  //    if so, then the value is retrieved and appended with the
  //        current location
  //    if not, then it is added as a new entry into the hashmap
  // get(), returns the <value> associated with the specified
  //    <key = recordID>
  if (map.get(recordID) != null) { // a <key, value> exists in
                                   //    the hashmap

    // create a new Text variable comprised of the retrieved
    //    <value> from the hashmap appended with the current
    //    values of the loop iteration
    tempList = new Text(map.get(recordID).toString()+
                        recordLoc.toString() + ", ").toString();

    // place the updated <value> in the hashmap assigned to the
    //    associated <key>
    // put(), places the specified <value> with the <key> in the
    //    hashmap, if the map already has a value associated with
    //    the key, it is replaced
    map.put(recordID, new Text(tempList));
  } // end if (<key, value> exists in hashmap)
  else { // the <key, value> does not exist in the hashmap

    // place a new entry in the hashmap
    map.put(recordID, new Text(recordLoc+", "));
  } // end else (<key, value> does not exist in hashmap)
} // end for ()

// create it, an Iterator that will be used to traverse and
```

```
    //    manipulate the HashMap
    // entrySet(), returns a Set view of the mappings contained in
    //    the HashMap
    // iterator(), returns an iterator over the set of elements
    Iterator<Entry<Text, Text>> it = map.entrySet().iterator();

    // create strBuilder, a StringBuilder is a mutable sequence of
    //    characters
    StringBuilder strBuilder = new StringBuilder();

    // iterate through all elements contained within the Iterator
    // hasNext(), returns TRUE is the iteration has more elements
    while (it.hasNext()) {
      // create pair, containing the map entry retrieved from the
      //    iterator
      // next(), returns the next element in the iteration
      Entry<Text, Text> pair = it.next();

      // append the StringBuilder with the <key, value> contained
      //    within pair
      // getKey(), returns the corresponding key of the entry
      // getValue(), returns the corresponding value of the entry
      strBuilder.append(pair.getKey() + ":" + pair.getValue() + " ");

      // remove already retrieved entry from the iterator
      // remove(), removes the last element returned by the iterator
      it.remove(); // avoids a ConcurrentModificationException }
    } // end while ()

    // send the final key value pair to output collector
    // key value pair is <key, strBuilder>
    // write() generates an output key value pair
    context.write(key, new Text(strBuilder.toString()));
  } // end reduce ()
} // end WETIndexLocationCustomReducer

public static class WETIndexLocationCustomPartitioner extends
                      Partitioner<Text, termLocationCustomWritable> {

  @Override
  public int getPartition(Text key, termLocationCustomWritable value,
                                              int numPartitions) {
  /*
   * getPartition is the method that returns the partition number for
   *    a given key
   */

    // create partitionKey, a String assigned to the value of the
    //    provided key
    String partitionKey = key.toString().toLowerCase();

    // this series of if-else statements checks the first character
    //    of the provided key and assigns the partitioner based on
    //    this value the purpose is to force the reducer output files
    //    to be organized by first character (e.g., all terms
//    beginning with 'a' are contained within the same output
//    file)
if (partitionKey.charAt(0) == '0')
  return 0;
else if (partitionKey.charAt(0) == '1')
  return 1;
else if (partitionKey.charAt(0) == '2')
  return 2;
else if (partitionKey.charAt(0) == '3')
  return 3;
else if (partitionKey.charAt(0) == '4')
  return 4;
else if (partitionKey.charAt(0) == '5')
  return 5;
else if (partitionKey.charAt(0) == '6')
  return 6;
else if (partitionKey.charAt(0) == '7')
  return 7;
else if (partitionKey.charAt(0) == '8')
  return 8;
else if (partitionKey.charAt(0) == '9')
  return 9;
else if (partitionKey.charAt(0) == 'a')
  return 10;
else if (partitionKey.charAt(0) == 'b')
  return 11;
else if (partitionKey.charAt(0) == 'c')
  return 12;
else if (partitionKey.charAt(0) == 'd')
  return 13;
else if (partitionKey.charAt(0) == 'e')
  return 14;
else if (partitionKey.charAt(0) == 'f')
  return 15;
else if (partitionKey.charAt(0) == 'g')
  return 16;
else if (partitionKey.charAt(0) == 'h')
  return 17;
else if (partitionKey.charAt(0) == 'i')
  return 18;
else if (partitionKey.charAt(0) == 'j')
  return 19;
else if (partitionKey.charAt(0) == 'k')
  return 20;
else if (partitionKey.charAt(0) == 'l')
  return 21;
else if (partitionKey.charAt(0) == 'm')
  return 22;
else if (partitionKey.charAt(0) == 'n')
  return 23;
else if (partitionKey.charAt(0) == 'o')
  return 24;
else if (partitionKey.charAt(0) == 'p')
  return 25;
else if (partitionKey.charAt(0) == 'q')
```

```
            return 26;
          else if (partitionKey.charAt(0) == 'r')
            return 27;
          else if (partitionKey.charAt(0) == 's')
            return 28;
          else if (partitionKey.charAt(0) == 't')
            return 29;
          else if (partitionKey.charAt(0) == 'u')
            return 30;
          else if (partitionKey.charAt(0) == 'v')
            return 31;
          else if (partitionKey.charAt(0) == 'w')
            return 32;
          else if (partitionKey.charAt(0) == 'x')
            return 33;
          else if (partitionKey.charAt(0) == 'y')
            return 34;
          else if (partitionKey.charAt(0) == 'z')
            return 35;
          else // this partition is the catch all for any unspecified
               //    character
            return 36;
        } // end getPartition ()
      } // end WETIndexLocationCustomPartitioner
    } // end WETIndexLocationCustomMap
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E. CUSTOM TUPLE-DATA TYPE

```java
package edu.nps.thesisProject.projectfiles;

/**
 * File:    termLocationCustomWritable.java
 * Purpose: Implementation of a custom writable data type.
 *
 * termLocationCustomWritable:
 *    Text         recordID   - contains the URL of the associated term
 *    IntWritable  location   - contains the location of a term
 *                              contained within the record
 *
 * @author adcoudra1@nps.edu
 */

// Java Packages
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

// Hadoop Packages
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;

public class termLocationCustomWritable implements Writable {

  // create the two elements of this custom writable datatype
  // recordID, a Text element containing a filename
  // location, an IntWritable containing the location of a term in the
  //    associated record
  private Text recordID;
  private IntWritable location;

  public termLocationCustomWritable() {
  /*
   * termLocationCustomWritable() is the default constructor for this
   *    custom data type
   */

    // create recordID, a new Text element
    this.recordID = new Text();

    // create location, a new IntWritable element
    this.location = new IntWritable();
  } // end termLocationCustomWritable constructor

  public void set(String recID, int loc) {
  /*
   * setter for the custom data type
   */
```

```
    // set recordID to input value of recID
    this.recordID.set(recID);

    // set location to input value of loc
    this.location.set(loc);
  } // end of set ()

  @Override
  public void readFields(DataInput dataInput) throws IOException {
  /*
   * readFields() deserializes the fields of the object from dataInput
   */

    recordID.readFields(dataInput);
    location.readFields(dataInput);
  } // end of readFields ()

  @Override
  public void write(DataOutput dataOutput) throws IOException {
  /*
   * write() serializes the fields of the object to dataOutput
   */

    recordID.write(dataOutput);
    location.write(dataOutput);
  } // end of write ()

  public Text getRecordID() {
  /*
   * getRecordID() returns the recordID of the custom data type
   *    variable
   */

    return recordID;
  } // end of getRecordID ()

  public IntWritable getLocation() {
  /*
   * getLocation() returns the location of the custom data type
   *    variable
   */

    return location;
  } // end of getLocation ()
} // end of termLocationCustomWritable class
```

APPENDIX F. WARC FILE WORD COUNT

```
package edu.nps.thesisProject.projectfiles;

/**
 * File:    WARCWordCount.java
 * Purpose: This file provides for a tool runner implementation of a
 *   MapReduce job.  The function performed is that of conducting a
 *   word count.  The expected input is in the form of WARC record
 *   entries from a WARC file.
 *
 * Driver for: WARCWordCountMap.java
 *
 * Input:   WARC files
 * Output:  <word> <count>
 *
 * @author adcoudra1@nps.edu
 */

// WARC Utilities
import edu.nps.thesisProject.warcutils.*;

// Hadoop Packages
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.lib.reduce.LongSumReducer;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

// Log4j Package
import org.apache.log4j.Logger;

public class WARCWordCount extends Configured implements Tool {

  // create a Logger object called LOG that retrieves a logger named
  //    after the class
  private static final Logger LOG =
                                Logger.getLogger(WARCWordCount.class);

  public static void main(String[] args) throws Exception {
  /*
   * main is the overall driver method that triggers the mapreduce job
   *    in hadoop
   */

    // RUN(CONFIGURATION conf, TOOL tool, STRING[] args)
    // runs the given Tool by Tool.run(String[]) after parsing given
    //    generic arguments
    // RUN(STRING[] args)
    // executes the command given the arguments
    //
    // overall this is equivalent to creating a new job and assigning
    //    the input and output path
    //
    // returns the exit code of the Tool.run(String[]) method
    int res = ToolRunner.run(new Configuration(),
                                      new WARCWordCount(), args);

    // terminates the currently running java virtual machine
    System.exit(res);
  } // end of main ()

  @Override
  public int run(String[] args) throws Exception {
  /*
   * Builds and runs the Hadoop job.
   * return 0 if the Hadoop job completes successfully and 1 otherwise.
   */

    // set the input and output path for the job
    // To take specified file paths from user, use:
    //            String inputPath = args[0];
    //            String outputPath = args[1];
    //
    // To take predefined file paths, use:
    //            String inputPath = "data/warc/*.warc.gz";
    //            String outputPath = "output/";
    //
    // Example paths for AWS:
    //    String inputPath = "s3n://aws-publicdatasets/common-
    //    crawl/crawl-data/CC-MAIN-2013-48/segments/1386163035819/wet/
    //    CC-MAIN-20131204131715-00000-ip-10-33-133-
    //    15.ec2.internal.warc.gz";
    //
    //    String inputPath = "s3n://aws-publicdatasets/common-
    //    crawl/crawl-data/CC-MAIN-2013-48/segments/
    //    1386163035819/wet/*.warc.gz";
    String inputPath = args[0];
    String outputPath = args[1];

    // Configuration processed by ToolRunner
    // getConf() returns the configuration used by this object
    Configuration conf = this.getConf();

    // create a job and assign a name (warcwordcount) for
    //    identification
    // getInstance(Configuration conf, String jobName) creates a job
    //    with the specified configuration and job name
    Job job = Job.getInstance(conf, "warcwordcount");

    // configure the InputFormat for the job as WarcInputFormat
    // setInputFormatClass() sets the InputFormat
    job.setInputFormatClass(WarcInputFormat.class);

    // configure the OutputFormat for the job as TextOutputFormat
    // setOutputFormatClass() sets the OutputFormat
    job.setOutputFormatClass(TextOutputFormat.class);

    // configure the key and value output class for the job
    // setOutputKeyClass() sets the key class for the job output data
    // setOutputValueClass() sets the value class for the job output
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(LongWritable.class);

    // configure the mapper, combiner (optional) and reducer classes
    // setMapperClass sets the mapper for the job
    // setCombinerClass sets the combiner class for the job
    // setReducerClass sets the reducer class for the job
    // provide mapper, combiner, and reducer class names
    job.setMapperClass(WARCWordCountMap.WARCWordCountMapper.class);
    job.setCombinerClass(LongSumReducer.class);
    job.setReducerClass(LongSumReducer.class);

    // configure the hdfs input and output directory as fetched from
    //    the command line
    // Path names a file or directory based on the argument string
    // setInputPaths sets the array of Path as the list of inputs for
    //    the job
    // setOutputPath sets the array of Path as the output directory for
    //    the job
    FileInputFormat.addInputPath(job, new Path(inputPath));
    FileOutputFormat.setOutputPath(job, new Path(outputPath));

    // configure the jar class
    // setJarByClass() sets the jar by finding where a given class came
    //    from
    job.setJarByClass(WARCWordCount.class);

    // log a message object with the info level (highlight progress of
    //    the application)
    LOG.info("Input path: " + inputPath);

    // submit the job to mapreduce and wait for completion, then return
    //    status
    // waitForCompletion() submits the job to the cluster and waits for
    //    it to finish
    return job.waitForCompletion(true) ? 0 : 1;
  } // end of run ()
} // end of WARCWordCount
```

```java
package edu.nps.thesisProject.projectfiles;

/**
 * File:    WARCWordCountMap.java
 * Purpose: This file provides a MapReduce job that performs the
 *   function of a word count on WARC record entries from a WARC file.
 *   This job only conducts a word count on 'response' type record
 *   entries with an html payload.
 *
 * Map for: WARCWordCount.java
 *
 * Input:   WARC files
 * Output:  <word1> <count>
 *          <word2> <count>
 *
 * @author adcoudra1@nps.edu
 */

// Java Packages
import java.io.IOException;
import java.util.StringTokenizer;

// Hadoop Packages
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

// Other Apache Packages
import org.apache.commons.io.IOUtils;
import org.apache.log4j.Logger;

// JWAT Packages
import org.jwat.common.Payload;
import org.jwat.warc.WarcRecord;

// Jsoup Packages
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;

public class WARCWordCountMap {

  // create a Logger object called LOG that retrieves a logger named
  //    after the class
  private static final Logger LOG =
                      Logger.getLogger(WARCWordCountMapper.class);

  // enum is a special data type that enables for a variable to be a
  //    set of predefined constants
  protected static enum MAPPERCOUNTER {

    EMPTY_RESPONSE,
    EXCEPTIONS,
    CURRENT_RECORD,
    RESPONSE_RECORDS,
    HTML_RECORDS
```

```
} // end MAPPERCOUNTER

protected static class WARCWordCountMapper extends
          Mapper<LongWritable, WarcRecord, Text, LongWritable> {

  // create a string tokenizer for the specified string line
  private StringTokenizer tokenizer;

  // create outKey, the eventual map output key
  private Text outKey = new Text();

  // create outVal, the output value equal to 1
  private LongWritable outVal = new LongWritable(1);

  @Override
  public void map(LongWritable key, WarcRecord value,
          Context context) throws IOException, InterruptedException {
  /*
   * map is the method that performs a tokenizer job, processing one
   *    record at a time
   */

    // setStatus(String msg) sets the current status of the task to
    //    the given string
    context.setStatus(MAPPERCOUNTER.CURRENT_RECORD + ": " +
                                                          key.get());

    // try following and if an error occurs, catch exception
    try {

      // only process records that are a 'response' type
      // value.header.warcTypeStr retrieves the "Type:" field of the
      //    record header
      if ("response".equals(value.header.warcTypeStr)) {

        // getCounter(Enum<?> counterName) gets the Counter
        //            for the given counterName
        // increment(long incr) increments the counter by the
        //            given value
        context.getCounter(MAPPERCOUNTER.RESPONSE_RECORDS).
                                                    increment(1);

        // Get the record payload
        // getPayload() returns the payload object for the record
        Payload payload = value.getPayload();

        // if payload is empty perform no operation
        // else process record for word count
        if (payload == null) {
          // NOP
        } // end if (payload is null)
        else {
          // create warcContent string from record payload
          // toString() converts the text (value) into a String
          // getInputStreamComplete() gets the InputStream to read
          //    the complete payload
          String warcContent =
                IOUtils.toString(payload.getInputStreamComplete());

          // if the record content is empty, increment counter
          // else process record content
          if (warcContent == null && "".equals(warcContent)) {

          context.getCounter(MAPPERCOUNTER.EMPTY_RESPONSE).
                                                    increment(1);
          } // end if (warcContent is empty)
          else {

            // create doc a Jsoup HTML Document
            // parse() parses HTML into a Document
            Document doc = Jsoup.parse(warcContent);

            // create html, a CharSequence assigned to the value of
            //    "html"
            CharSequence html = "html";

            // create metaInfo, a String used to retrieve the content
            //    type of the payload
            // select(String cssQuery) finds elements matching the
            //    provided selector
            // first() returns the first matched element
            // attr(String) gets an attributes value by its key
            String metaInfo = doc.select(
                                  "meta[http-equiv=Content-Type]").
                                        first().attr("content");

            // if the payload is coded in html, continue evaluating
            // else process the next record
            if (metaInfo.contains(html)) {

              context.getCounter(MAPPERCOUNTER.HTML_RESPONSE).
                                                    increment(1);

              // create text, a String that contains the text of the
              //    HTML Document body
              // body() accesses the Documents body element
              // text() gets the body element as text
              String text = doc.body().text();

              // assign the warcConent to string tokenizer
              tokenizer = new StringTokenizer(text);

              // iterate through all words (tokens) in the record
              //    body take next token and form key value pair
              // hasMoreTokens() tests for more tokens in the
              //    tokenizer’s string and returns a boolean
              while (tokenizer.hasMoreTokens()) {

                // set the outKey as the token
                outKey.set(tokenizer.nextToken());
```

```
                  // send key value pair to output collector to pass to
                  //    the reducer
                  // key value pair is <Text outKey, 1>
                  // write() generates an output key value pair
                  context.write(outKey, outVal);
                } // end while (more tokens)
              } // end if (html content)
              else { // if response content is not html
                // NOP
              } // end else (not html)
            } // end else (warcContent has contents)
          } // end else (payload has contents)
        } // end if (record is response)
        else { // if record is not a response
          // NOP
        } // end else (record not a response)
      } // end try ()
      catch (Exception ex) { // if the try statement fails,
                             //    catch the exception

        // error(String msg, Throwable t)
        // output the internal error statement
        LOG.error("Caught Exception", ex);

        // when exception occurs, increment the counter
        context.getCounter(MAPPERCOUNTER.EXCEPTIONS).increment(1);
      } // end catch ()
    } // end map ()
  } // end WARCWordCountMapper
} // end WARCWordCountMap
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX G. WARC FILE INVERTED INDEX

```
package edu.nps.thesisProject.projectfiles;

/**
 * File:    WARCIndexLocationCustom.java
 * Purpose: This file provides for a tool runner implementation of a
 *   MapReduce job.  The function performed is that a creating an
 *   inverted index using a custom tuple data type.  The expected input
 *   is in the form of WARC record entries from a WARC file.
 *
 * Driver for: WARCIndexLocationCustomMap.java
 *
 * Input:   WARC files
 * Output:  <word1> | <url1>|<location1>, <location2>,|
                      <url2>|<location1>, <location2>,|
            <word2> | <url1>|<location1>, <location2>,|
                      <url2>|<location1>, <location2>,|
 *
 * @author adcoudra1@nps.edu
 */

//WARC Utilities
import edu.nps.thesisProject.warcutils.*;

// Hadoop Packages
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

//Log4j Package
import org.apache.log4j.Logger;

public class WARCIndexLocationCustom extends Configured implements
                                                            Tool {

  // create a Logger object called LOG that retrieves a logger named
  //    after the class
  private static final Logger LOG =
                Logger.getLogger(WARCIndexLocationCustom.class);

  public static void main(String[] args) throws Exception {
  /*
   * main is the overall driver method that triggers the mapreduce job
   *    in hadoop
   */
  // RUN(CONFIGURATION conf, TOOL tool, STRING[] args)
  // runs the given Tool by Tool.run(String[]) after parsing given
  //    generic arguments
  // RUN(STRING[] args)
  // executes the command given the arguments
  //
  // overall this is equivalent to creating a new job and assigning
  //    the input and output path
  //
  // returns the exit code of the Tool.run(String[]) method
  int res = ToolRunner.run(new Configuration(),
                          new WARCIndexLocationCustom(), args);

  // terminates the currently running java virtual machine
  System.exit(res);
} // end main ()

@Override
public int run(String[] args) throws Exception {
/*
 * Builds and runs the Hadoop job.
 * return 0 if the Hadoop job completes successfully and 1 otherwise.
 */

  // set the input and output path for the job
  // To take specified file paths from user, use:
  //      String inputPath = args[0];
  //      String outputPath = args[1];
  //
  // To take predefined file paths, use:
  //      When all files are contained within a single directory:
  //            String inputPath = "data/warc/*.warc.wet.gz";
  //
  //      When input files are contained with multiple directions use
  //          globbing:
  //      Example:
  //            data---warc---test1---file1.warc.gz
  //                 |      |
  //                 |      |-test2---file2.warc.gz
  //                 |
  //                 |-wet----test1---file1.warc.wet.gz
  //                        |
  //                        |-test2---file2.warc.wet.gz
  //
  //      To search entire structure and identify all warc.gz files:
  //            String intputPath = "data/*/*/*.warc.gz";
  //
  //            String outputPath = "output/";
  //
  // Example paths for AWS:
  //      String inputPath = "s3n://aws-publicdatasets/common-
  //          crawl/crawl-data/CC-MAIN-2013-
  //          48/segments/1386163035819/wet/CC-MAIN-20131204131715-
  //          00000-ip-10-33-133-15.ec2.internal.warc.gz";
```

```
//
//      String inputPath = "s3n://aws-publicdatasets/common-
//          crawl/crawl-data/CC-MAIN-2013-
//          48/segments/1386163035819/wet/*.warc.gz";
String inputPath = args[0];
String outputPath = args[1];

// Configuration processed by ToolRunner
// getConf() returns the configuration used by this object
Configuration conf = this.getConf();

// Compress the final reducer output
// set(String name, String value) sets the value of the named
//    property
// mapreduce.output.fileoutputformat.compress
//      default: false
//      purpose: specify whether or no job output should be
//          compressed
// mapreduce.output.fileoutputformat.compress.codec
//      default: org.apache.hadoop.io.compress.DefaultCodec
//      purpose: specify compression codec
conf.set("mapreduce.output.fileoutputformat.compress", "true");
conf.set("mapreduce.output.fileoutputformat.compress.codec",
                    "org.apache.hadoop.io.compress.GzipCodec");

// create a job and assign a name (warcindexlocationcustom) for
//    identification
// getInstance(Configuration conf, String jobName) creates a job
//    with the specified configuration and job name
Job job = Job.getInstance(conf, "warcindexlocationcustom");

// configure the InputFormat for the job as WarcInputFormat
// setInputFormatClass() sets the InputFormat
job.setInputFormatClass(WarcInputFormat.class);

// configure the Map OutputValue for the job as
//    termLocationCustomWritable
// setMapOutputValueClass() set the MapOutputValue
job.setMapOutputValueClass(termLocationCustomWritable.class);

// configure the OutputFormat for the job as TextOutputFormat
// setOutputFormatClass() sets the OutputFormat
job.setOutputFormatClass(TextOutputFormat.class);

// configure the key and value output class for the job
// setOutputKeyClass() sets the key class for the job output data
// setOutputValueClass() sets the value class for the job output
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);

// configure the mapper, combiner (optional) and reducer classes
// setMapperClass sets the mapper for the job
// setCombinerClass sets the combiner class for the job
// setReducerClass sets the reducer class for the job
// provide mapper, combiner, and reducer class names
```

```
    job.setMapperClass(WARCIndexLocationCustomMap.
                                WARCIndexLocationCustomMapper.class);
    job.setReducerClass(WARCIndexLocationCustomMap.
                                WARCIndexLocationCustomReducer.class);

    // configure the partitions of the key space
    // setPartitionerClass() sets the partitions class for the job
    // setNumReduceTasks() sets the number of reduce tasks for the job
    // NOTE: if partitioner is used, then number of reduce tasks must
    //    match the number of partitions used
    job.setPartitionerClass(WARCIndexLocationCustomMap.
                          WARCIndexLocationCustomPartitioner.class);
    job.setNumReduceTasks(37);

    // configure the hdfs input and output directory as fetched from
    //    the command line
    // Path names a file or directory based on the argument string
    // setInputPaths sets the array of Path as the list of inputs for
    //    the job
    // setOutputPath sets the array of Path as the output directory for
    //    the job
    FileInputFormat.addInputPath(job, new Path(inputPath));
    FileOutputFormat.setOutputPath(job, new Path(outputPath));

    // configure the jar class
    // setJarByClass() sets the jar by finding where a given class came
    //    from
    job.setJarByClass(WARCIndexLocationCustom.class);

    // log a message object with the info level (highlight progress of
    //    the application)
    LOG.info("Input path: " + inputPath);

    // submit the job to mapreduce and wait for completion, then return
    //    status
    // waitForCompletion() submits the job to the cluster and waits for
    //    it to finish
    return job.waitForCompletion(true) ? 0 : 1;
  } // end run ()
} // WARCIndexCustomLocation
```

```
package edu.nps.thesisProject.projectfiles;

/**
 * File:    WARCIndexLocationCustomMap.java
 * Purpose: This file provides a MapReduce job that performs the
 *   function of an inverted index on WARC record entries from a WARC
 *   file, using a custom tuple data type.
 *
 * Map for: WARCIndexLocationCustom.java
 *
 * Input:   WARC files
 * Output:  <word1> | <url1>|<location1>, <location2>,|
 *                    <url2>|<location1>, <location2>,|
 *          <word2> | <url1>|<location1>, <location2>,|
 *                    <url2>|<location1>, <location2>,|
 *
 * @author adcoudra1@nps.edu
 */

// Java Packages
import java.io.IOException;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map.Entry;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

// Hadoop Packages
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;

// Other Apache Packages
import org.apache.commons.io.IOUtils;
import org.apache.log4j.Logger;

// JWAT Packages
import org.jwat.common.Payload;
import org.jwat.warc.WarcRecord;

//Jsoup Packages
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.select.Elements;

public class WARCIndexLocationCustomMap {

  // create a Logger object called LOG that retrieves a logger named
  //    after the class
  private static final Logger LOG =
            Logger.getLogger(WARCIndexLocationCustomMapper.class);
  // enum is a special data type that enables for a variable to be a
  //    set of predefined constants
  protected static enum MAPPERCOUNTER {

    EMPTY_RESPONSE,
    EXCEPTIONS,
    CURRENT_RECORD,
    RESPONSE_RECORDS,
    HTML_RECORDS
  } // end MAPPERCOUNTER

  protected static class WARCIndexLocationCustomMapper extends
                              Mapper<LongWritable, WarcRecord, Text,
                                          termLocationCustomWritable> {

    // NOTE: Two different regex patterns are offered – keep one
    //    commented out
    // regex method (1):
    // create pattern, a compiled representation of a regular
    //    expression
    // Pattern is the overarching class
    // compile() creates the pattern based on the provided flags
    // \w indicates a word character, short for [a-zA-Z_0-9]
    // \\w the first \ is the escape character so that Java will
    //    recognize '\w' as the input
    // + indicates occurring one or more times, short for {1,}
//    private static Pattern pattern = Pattern.compile("\\w+");

    // regex method (2):
    // (?=[a-zA-Z0-9]+[a-zA-Z]) is a positive lookahead assertion
    //  ?=                       : positive lookahead
    //    [a-zA-Z0-9]+           : match any characters in the list, an
    //                               unlimited number of times
    //                [a-zA-Z]   : that contains at least one of the
    //                             specified characters
    //    => a single character in the specified list [] must be
    //       present
    // ([a-zA-Z0-9]+) is the first capturing group
    //             +      : between one and unlimited times, as many
    //                      times as possible
    //   a-z              : a single character between lower case
    //                      a and z
    //      A-Z           : a single character between upper case
    //                      A and Z
    //         0-9        : a single character between 0 and 9
    //    => this regex will match any combination of letters and
    //       numbers, excluding only number combinations
    private static Pattern pattern =
            Pattern.compile("(?=[a-zA-Z0-9]+[a-zA-Z])([a-zA-Z0-9]+)");

    private termLocationCustomWritable recordLocation =
                                    new termLocationCustomWritable();

    // create commonWords, a list of strings for common words to
    //    exclude from counting process
    // Arrays is a class containing methods for manipulating arrays
    // asList() creates a fixed size list initialized as specified
    List<String> commonWords = Arrays.asList("a", "about", "am", "an",
                  "and", "are", "as", "at", "be", "by", "com", "de",
                  "en", "for", "from", "how", "i", "in", "is", "it",
                  "la", "not", "of", "on", "or", "that", "the", "this",
                  "to", "was", "what", "when", "where", "who", "will",
                  "with", "www");

    // create recordID, to be used to contain the current record ID
    private String recordID = new String();

    // create word, the eventual map output key
    private Text word = new Text();

    @Override
    public void map(LongWritable key, WarcRecord value,
            Context context) throws IOException, InterruptedException {
    /*
     * map is the method that performs a tokenizer job, processing one
     *    record at a time
     */

      // setStatus(String msg) sets the current status of the task to
      //    the given string
      context.setStatus(MAPPERCOUNTER.CURRENT_RECORD + ": " +
                                                            key.get());

      // try following and if an error occurs, catch exception
     try {

        // only process records that are a 'response' type
        // value.header.warcTypeStr retrieves the "Type:" field of the
        //    record header
       if ("response".equals(value.header.warcTypeStr)) {

          // getCounter(Enum<?> counterName) gets the Counter
          //    for the given counterName
          // increment(long incr) increments the counter by the
          //    given value
          context.getCounter(MAPPERCOUNTER.RESPONSE_RECORDS).
                                                      increment(1);

          // Get the record payload
          // getPayload() returns the payload object for the record
          Payload payload = value.getPayload();

          // if payload is empty perform no operation
          // else process record for word count
          if (payload == null) {
            // NOP
          } // end if (payload is null)
          else { // payload exists
            // create warcContent string from record payload
    // toString() converts the text (value) into a String
    // getInputStreamComplete() gets the InputStream to read
    //    the complete payload
    String warcContent =
          IOUtils.toString(payload.getInputStreamComplete());

    // if the record content is empty, increment counter
    // else process record content
    if (warcContent == null && "".equals(warcContent)) {

      context.getCounter(MAPPERCOUNTER.EMPTY_RESPONSE).
                                            increment(1);
    } // end if (payload is empty)
    else { // payload is not empty
      // create doc a Jsoup HTML Document
      // parse() parses HTML into a Document
      Document doc = Jsoup.parse(warcContent);

      // method (1): search for content-type
      // -> uncomment method (1) section
      // -> comment out method (2) section

      /* // <- start of method (1)
      // create html, a CharSequence assigned to the value of
      //    "html"
      CharSequence html = "html";

      // create metaInfo, a String used to retrieve the content
      //    type of the payload
      // select(String cssQuery) finds elements matching the
      //    provided selector
      // first() returns the first matched element
      // attr(String) gets an attributes value by its key
      String metaInfo = doc.select(
                          "meta[http-equiv=Content-Type]").
                                first().attr("content");
      */ // <- end of method (1)

      ///* // <- start of method (2)
      // create HTML, an object of the type Elements which
      //    contain the retrieved result from the jsoup
      //    Document
      // select("html") finds the contents of the <html> tag in
      //    the document this returns null if none present
      Elements span = doc.select("html");
      //*/ // <- end of method (2)

      // if the payload is coded in html, continue evaluating
      // else process the next record
      // if (metaInfo.contains(html)) { // <- if for method (1)
      if (span != null) {  // <- if statement for method (2)

        context.getCounter(MAPPERCOUNTER.HTML_RECORDS).
                                            increment(1);
```

```
// create text, a String that contains the text of the
//    HTML Document body
// body() accesses the Documents body element
// text() gets the body element as text
String text = doc.body().text();

// create matcher to perform regex operation on the
//    input String
// matcher() matches the input sequence against the
//    provided pattern
// toString() converts the text (value) into a String
Matcher matcher = pattern.matcher(text);

// obtain the url of the current record being processed
// value.header.warcTargetUriStr retrieves the
//    "WARC-Target-URI:" field of the record header
recordID = value.header.warcTargetUriStr;

// initialize wordLocation to 0
int wordLocation = 0;

// iterate through all matches of the regex pattern
//    found in the input take the next match, convert
//    to lower case and check against common words, if
//    not contained in common words, form and emit the
//    key value pair
// find() attempts to find the next subsequence of the
//    input sequence that matches
while(matcher.find()) {

  // increment wordLocation
  wordLocation++;

  // create matchedKey from each matched pattern
  //    converted to lower case
  // group() returns the input subsequence matched by
  //    the previous match
  // toLowerCase() converts the input to all lower case
  String matchedKey =
            matcher.group().trim().toLowerCase();

  // test to see if token is contained in commonWords,
  //    if so do not create key value pair
  // toLowerCase() converts all characters in string to
  //    lower case
  // trim() returns a copy of the string with all
  //    leading and trailing whitespace removed
  // contains() tests string for specified character
  //    sequence and returns a boolean
  if (!commonWords.contains(matchedKey)) {

    // set the value of word to the matched word,
    //    concatenated with "|"
    word.set(new Text(matchedKey+"|"));
```

```java
                  // set the value of the custom datatype
                  recordLocation.set(recordID, wordLocation);

                  // send key value pair to output collector to pass
                  //    to the reducer
                  // key value pair is <word, (url, location)>
                  // write() generates an output key value pair
                  context.write(word, recordLocation);
                } // end if (pattern not found in common words
              } // end while (pattern exists in body)
              // reset word counter
              wordLocation = 0;
            } // end if response content is html
            else { // if response content is not html
              // NOP
            } // end else (response not html)
          } // end else (payload not empty)
        } // end else (payload exists)
      } // end else (payload != null)
      else { // if record is not a response
        // NOP
      } // end else (not a response)
    } // end try ()
    catch (Exception ex) { // if try statement fails, catch exception
      // error(String msg, Throwable t)
      // output the internal error statement
      LOG.error("Caught Exception", ex);

      // when exception occurs, increment the counter
      context.getCounter(MAPPERCOUNTER.EXCEPTIONS).increment(1);
    } // end catch (exception)
  } // end map ()
} // end WARCIndexLocationCustomMapper

public static class WARCIndexLocationCustomReducer extends
          Reducer<Text, termLocationCustomWritable, Text, Text> {

  @Override
  public void reduce(Text key, Iterable<termLocationCustomWritable>
                                values, Context context) throws
                                IOException, InterruptedException {
  /*
   * reduce is the method that accepts key value pairs from the
   *    mappers, sums up the values for each key and produces a final
   *    output
   */

    // create map, a HashMap<K, V> of type <Text, Text>
    HashMap<Text, Text> map = new HashMap<Text, Text>();

    // create tempList, a String that will be used to hold a
    //    temporary hashmap value
    String tempList = "";

    // create val, of the custom type termLocationCustomWritable that
//    iterates through each termLocationCustomWritable value
// this loop iterates through each value passed for a given key
for (termLocationCustomWritable val : values) {

  // create recordID, and assign to it the RecordID (URL)
  //    associated with val
  // getRecordID() returns the Text recordID contained within the
  //    custom data type
  Text recordID = new Text(val.getRecordID());

  // create recordLoc, and assign to it the Location (word
  //    position) associated with val
  // getLocation() returns the IntWritable location contained
  //    within the custom data type
  // toString() converts the IntWritable to a String
  Text recordLoc = new Text(val.getLocation().toString());

  // check the map to see if it contains any values associated
  //    with the specified key this checks to see to see whether
  //    or not a key value has been placed in the hashmap
  //    -> if so, then the value is retrieved and appended with
  //       the current location
  //    -> if not, then it is added as a new entry into the
  //       hashmap
  // get(), returns the <value> associated with the specified
  //    <key = recordID>
  if (map.get(recordID) != null) { // a <key, value> exists in
                                    //    hashmap

    // create a new Text variable comprised of the retrieved
    //    <value> from the hashmap appended with the current
    //    values of the loop iteration
    tempList = new Text(map.get(recordID).toString()
                        +recordLoc.toString()+",").toString();

    // place the updated <value> in the hashmap assigned to the
    //    associated <key>
    // put(), places the specified <value> with the <key> in the
    //    hashmap, if the map already has a value associated with
    //    the key, it is replaced
    map.put(recordID, new Text(tempList));
  } // end if (<key, value> exists in hashmap)
  else { // the <key, value> does not exist in the hashmap

    // place a new entry in the hashmap
    map.put(recordID, new Text(recordLoc+","));
  } // end else (<key, value> not in hashmap)
} // end for (each termLocationCustomWritable)

// create it, an Iterator that will be used to traverse and
//    manipulate the HashMap
// entrySet(), returns a Set view of the mappings contained in
//    the HashMap
// iterator(), returns an iterator over the set of elements
Iterator<Entry<Text, Text>> it = map.entrySet().iterator();
```

```
    // create strBuilder, a StringBuilder is a mutable sequence of
    //    characters
    StringBuilder strBuilder = new StringBuilder();

    // iterate through all elements contained within the Iterator
    // hasNext(), returns TRUE is the iteration has more elements
    while (it.hasNext()) {

      // create pair, containing the map entry retrieved from the
      //    iterator
      // next(), returns the next element in the iteration
      Entry<Text, Text> pair = it.next();

      // append the StringBuilder with the <key, value> contained
      //    within pair
      // getKey(), returns the corresponding key of the entry
      // getValue(), returns the corresponding value of the entry
      strBuilder.append(pair.getKey() + "|" + pair.getValue() + "|");

      // remove already retrieved entry from the iterator
      // remove(), removes the last element returned by the iterator
      it.remove(); // avoids a ConcurrentModificationException }
    } // end while(more elements in iterator)

    // send the final key value pair to output collector
    // key value pair is <key, strBuilder>
    // write() generates an output key value pair
    context.write(key, new Text(strBuilder.toString()));
  } // end reduce ()
} // end WARCIndexCustomLocationReducer

public static class WARCIndexLocationCustomPartitioner extends
                         Partitioner<Text, termLocationCustomWritable> {

  @Override
  public int getPartition(Text key, termLocationCustomWritable value,
                                                  int numPartitions) {
  /*
   * getPartition is the method that returns the partition number for
   *    a given key
   */

    // create partitionKey, a String assigned to the value of the
    //    provided key
    String partitionKey = key.toString().toLowerCase();

    // this series of if-else statements checks the first character
    //    of the provided key and assigns the partitioner based on
    //    this value the purpose is to force the reducer output files
    //    to be organized by first character (e.g., all terms
    //    beginning with 'a' are contained within the same output
    //    file)
    if (partitionKey.charAt(0) == '0')
      return 0;
```

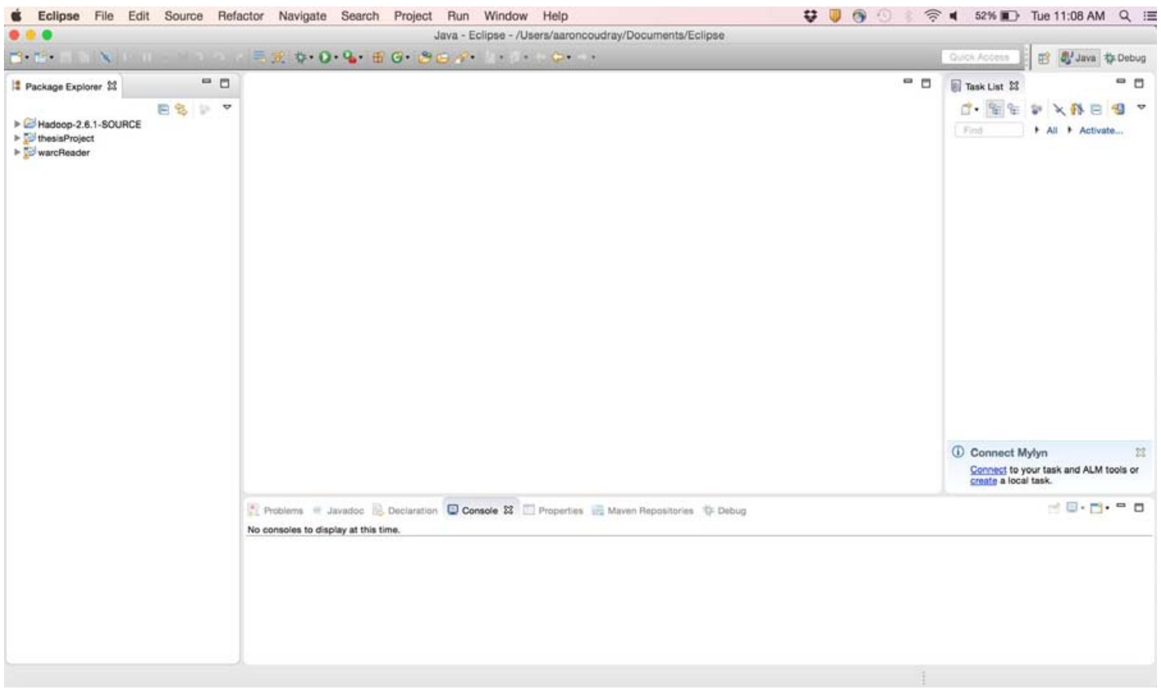
```
else if (partitionKey.charAt(0) == '1')
  return 1;
else if (partitionKey.charAt(0) == '2')
  return 2;
else if (partitionKey.charAt(0) == '3')
  return 3;
else if (partitionKey.charAt(0) == '4')
  return 4;
else if (partitionKey.charAt(0) == '5')
  return 5;
else if (partitionKey.charAt(0) == '6')
  return 6;
else if (partitionKey.charAt(0) == '7')
  return 7;
else if (partitionKey.charAt(0) == '8')
  return 8;
else if (partitionKey.charAt(0) == '9')
  return 9;
else if (partitionKey.charAt(0) == 'a')
  return 10;
else if (partitionKey.charAt(0) == 'b')
  return 11;
else if (partitionKey.charAt(0) == 'c')
  return 12;
else if (partitionKey.charAt(0) == 'd')
  return 13;
else if (partitionKey.charAt(0) == 'e')
  return 14;
else if (partitionKey.charAt(0) == 'f')
  return 15;
else if (partitionKey.charAt(0) == 'g')
  return 16;
else if (partitionKey.charAt(0) == 'h')
  return 17;
else if (partitionKey.charAt(0) == 'i')
  return 18;
else if (partitionKey.charAt(0) == 'j')
  return 19;
else if (partitionKey.charAt(0) == 'k')
  return 20;
else if (partitionKey.charAt(0) == 'l')
  return 21;
else if (partitionKey.charAt(0) == 'm')
  return 22;
else if (partitionKey.charAt(0) == 'n')
  return 23;
else if (partitionKey.charAt(0) == 'o')
  return 24;
else if (partitionKey.charAt(0) == 'p')
  return 25;
else if (partitionKey.charAt(0) == 'q')
  return 26;
else if (partitionKey.charAt(0) == 'r')
  return 27;
else if (partitionKey.charAt(0) == 's')
```

```
        return 28;
      else if (partitionKey.charAt(0) == 't')
        return 29;
      else if (partitionKey.charAt(0) == 'u')
        return 30;
      else if (partitionKey.charAt(0) == 'v')
        return 31;
      else if (partitionKey.charAt(0) == 'w')
        return 32;
      else if (partitionKey.charAt(0) == 'x')
        return 33;
      else if (partitionKey.charAt(0) == 'y')
        return 34;
      else if (partitionKey.charAt(0) == 'z')
        return 35;
      else // this partition is the catch all for any unspecified
           //    character
        return 36;
    } // end getPartition ()
  } // end WARCIndexLocationCustomPartitioner
} // end WARCIndexLocationCustomMap
```

APPENDIX H. PROJECT INSTALLATION INSTRUCTIONS

In order to generate functioning MapReduce jobs from the source code contained within this thesis, or provided as a part of the supplemental material, the entirety of the project files must be properly loaded. The following steps provide instructions on how to load this thesis as a Maven project into Eclipse:

1. Open Eclipse (see Figure 14).



Connect to your task and ALM tools or create a local task.

No consoles to display at this time.

Figure 14. Screen Shot of Eclipse

2. Select: File -> Import (see Figure 15).

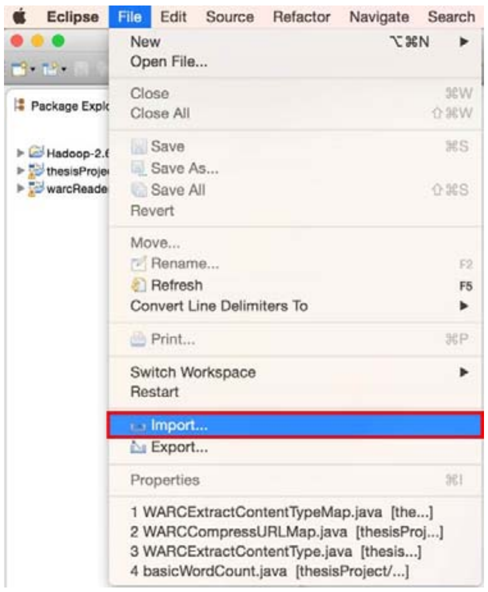

Eclipse File Edit Source Refactor Navigate Search
Package Explorer
Hadoop-2.
thesisProje
warcReade
New ⌥⌘N
Open File...
Close ⌘W
Close All ⇧⌘W
Save ⌘S
Save As...
Save All ⇧⌘S
Revert
Move...
Rename... F2
Refresh F5
Convert Line Delimiters To
Print... ⌘P
Switch Workspace
Restart
Import...
Export...
Properties ⌘I
1 WARCExtractContentTypeMap.java [the...]
2 WARCCompressURLMap.java [thesisProj...]
3 WARCExtractContentType.java [thesis...]
4 basicWordCount.java [thesisProject/...]

Figure 15. Screen Shot of Import Action

3. Select: Existing Maven Projects -> Next (see Figure 16).

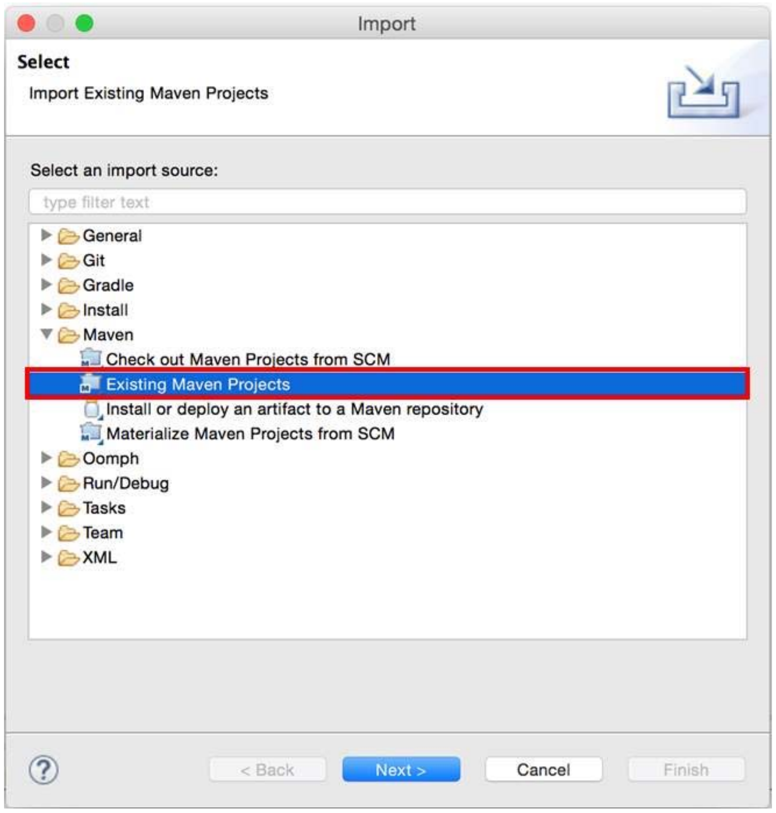


Figure 16. Screen Shot of Selecting Existing Maven Project

4. Select: Browse to Root Directory (see Figure 17).

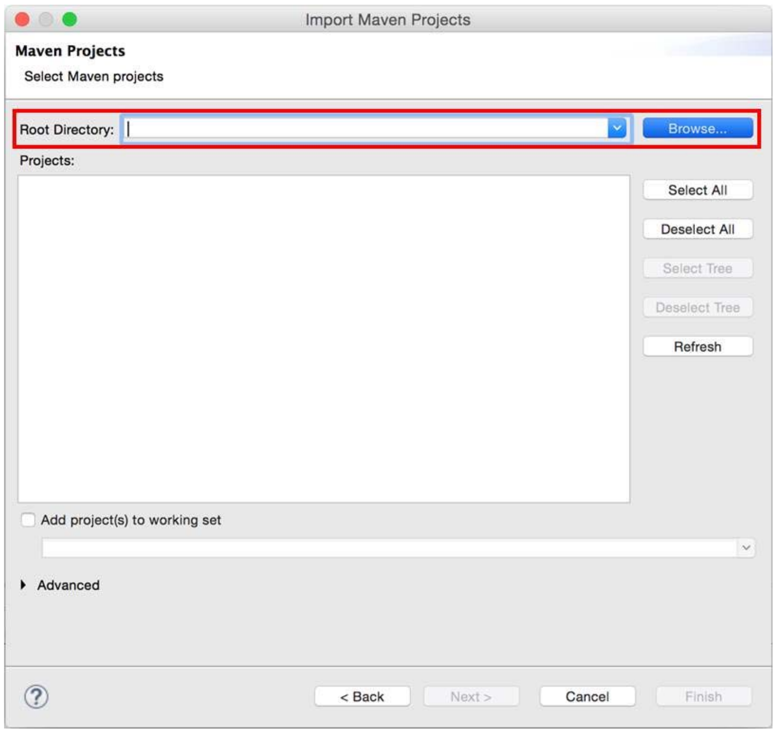


Figure 17. Screen Shot of Selecting Browse to Root Directory

5. Select: Root Folder -> Open (see Figure 18).

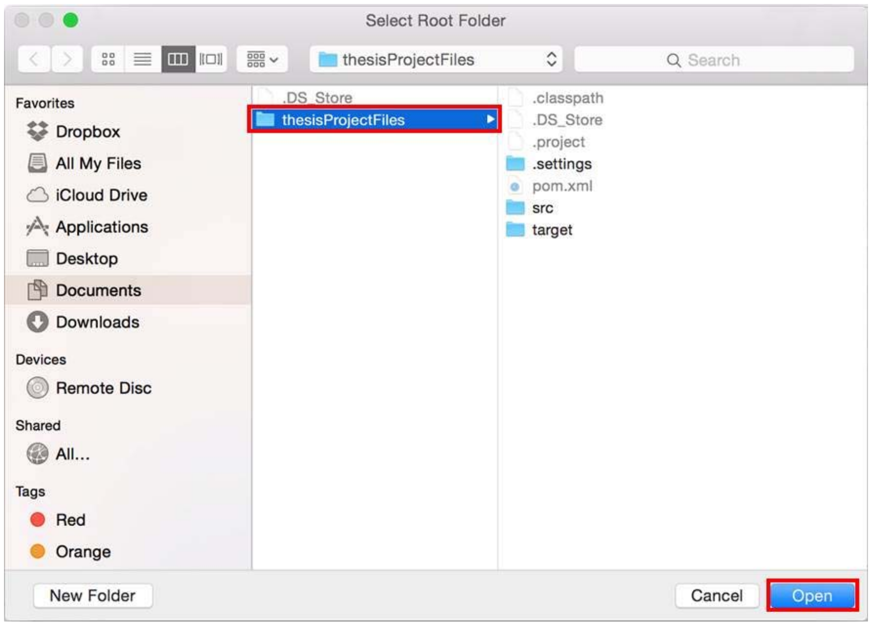
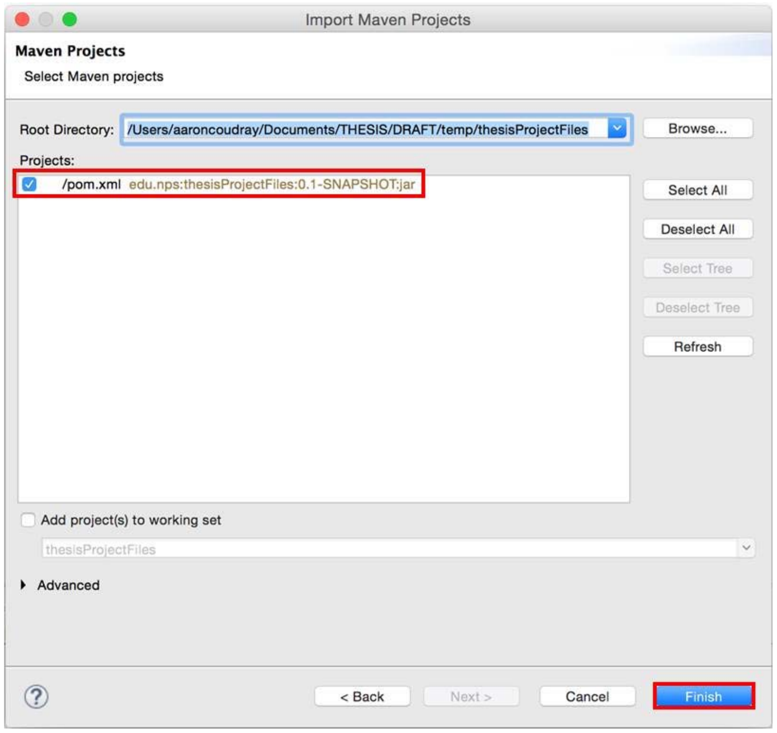


Figure 18. Screen Shot of Selecting Project Root Folder

6. Ensure pom.xml file is checked and select Finish (see Figure 19).



Root Directory: /Users/aaroncoudray/Documents/THESIS/DRAFT/temp/thesisProjectFiles

Projects:

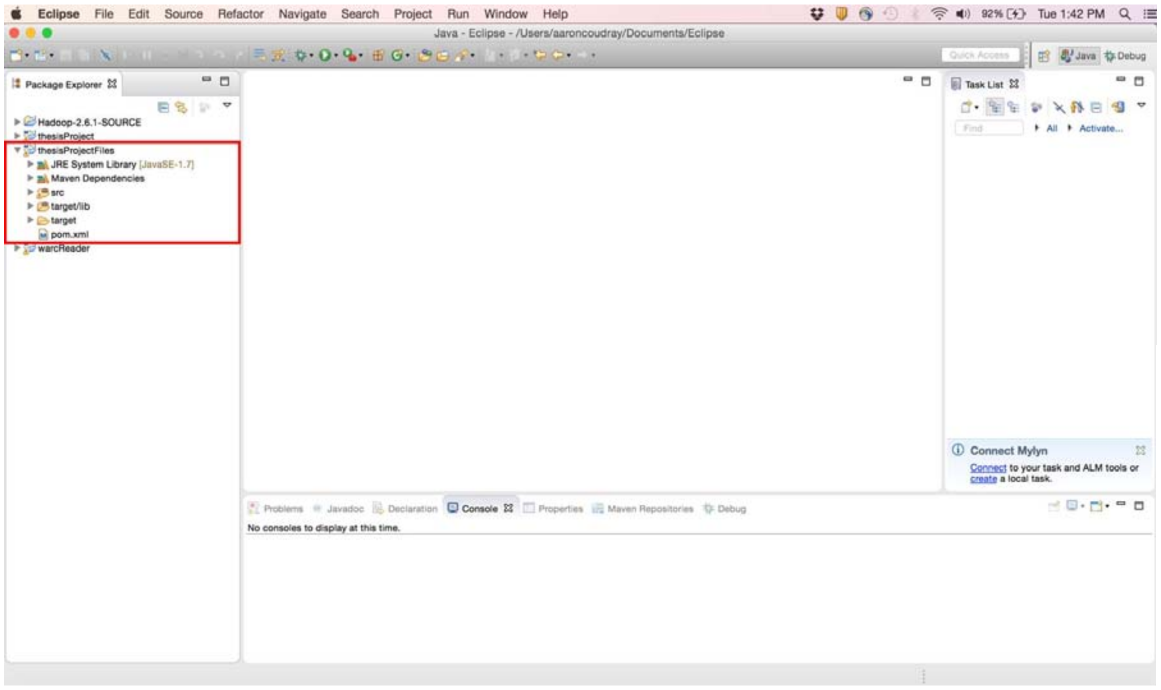
☑ /pom.xml edu.nps:thesisProjectFiles:0.1-SNAPSHOT:jar

☐ Add project(s) to working set

thesisProjectFiles

Figure 19. Screen Shot of Selecting POM File and Finishing Import

7. This completes the import process of an existing Maven project (see Figure 20).


Eclipse File Edit Source Refactor Navigate Search Project Run Window Help
Java - Eclipse - /Users/aaroncoudray/Documents/Eclipse
Package Explorer
Hadoop-2.6.1-SOURCE
thesisProject
thesisProjectFiles
JRE System Library [JavaSE-1.7]
Maven Dependencies
src
target/lib
target
pom.xml
warcReader
Task List
Connect Mylyn
Connect to your task and ALM tools or create a local task.
Problems Javadoc Declaration Console Properties Maven Repositories Debug

No consoles to display at this time.

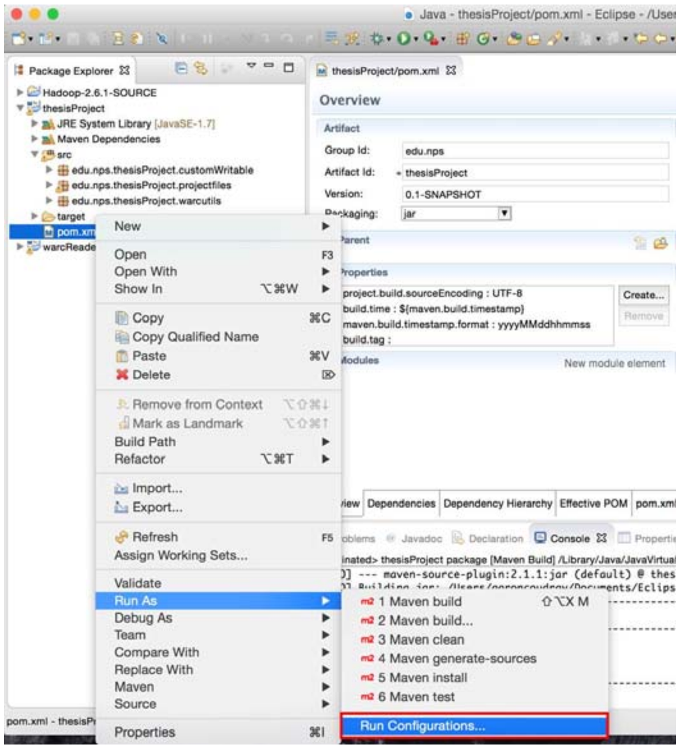
Figure 20. Screen Shot of Project Loaded into Eclipse

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX I. PROJECT JAR CREATION INSTRUCTIONS

In order to run a MapReduce job from source code contained within this thesis, or provided as a part of the supplemental material, the entirety of the project files must be properly packaged into a JAR file. The following steps provide instructions on how to create a JAR file using the Maven plugin as a part of Eclipse:

1. Right click on the pom.xml, then select: Run As -> Run Configuration (see Figure 21).



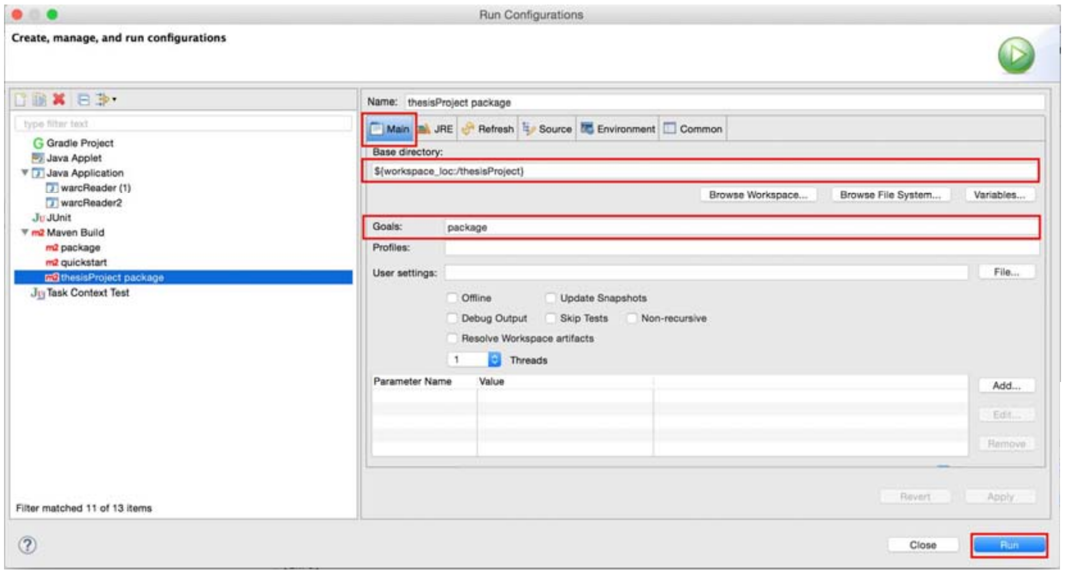
Group Id: edu.nps
Artifact Id: thesisProject
Version: 0.1-SNAPSHOT
Packaging: jar

project.build.sourceEncoding : UTF-8
build.time : ${maven.build.timestamp}
maven.build.timestamp.format : yyyyMMddhhmmss
build.tag :

```
inated> thesisProject package [Maven Build] /Library/Java/JavaVirtual
O] --- maven-source-plugin:2.1.1:jar (default) @ thes
```

Figure 21.   Screen Shot of Selecting Run Configuration

2. Create a custom Maven Build. On the Main tab: (1) set the Base Directory for the project and (2) set the Goal to “package” (see Figure 22).


Run Configurations

Create, manage, and run configurations

Name: thesisProject package

Base directory: ${workspace_loc:/thesisProject}

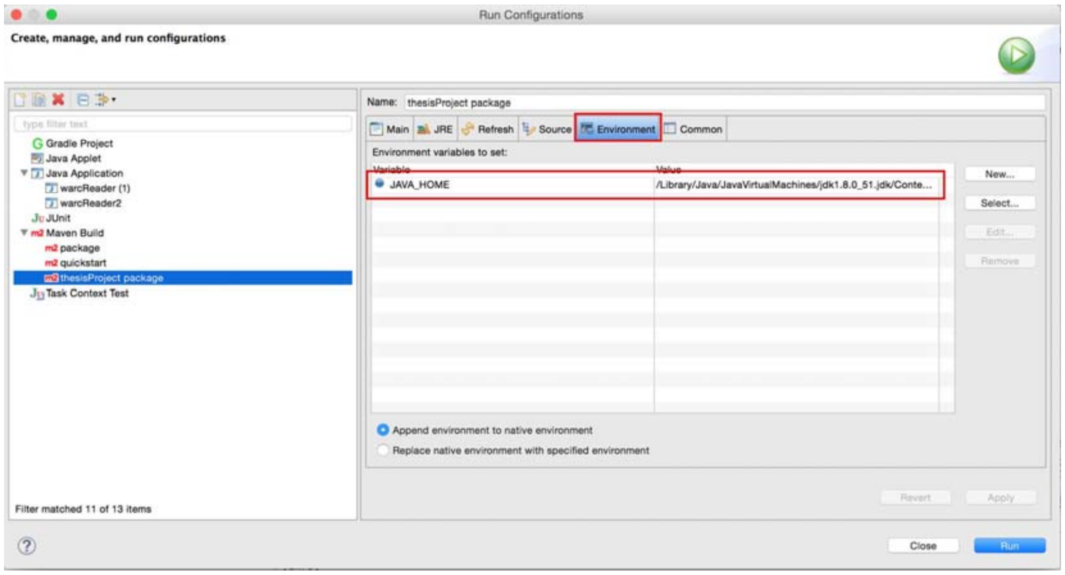
Goals: package

Threads: 1

Filter matched 11 of 13 items

Figure 22. Screen Shot of Main Tab Actions for Creating a Custom Maven Build

3. Create a custom Maven Build. On the Environment tab: (3) create a “JAVA_HOME” variable and set the value according the location of JAVA_HOME on the computer in use (see Figure 23). This step is not required for successful JAR creation.


Run Configurations

Create, manage, and run configurations

Name: thesisProject package

Environment variables to set:

Variable	Value
JAVA_HOME	/Library/Java/JavaVirtualMachines/jdk1.8.0_51.jdk/Conte...

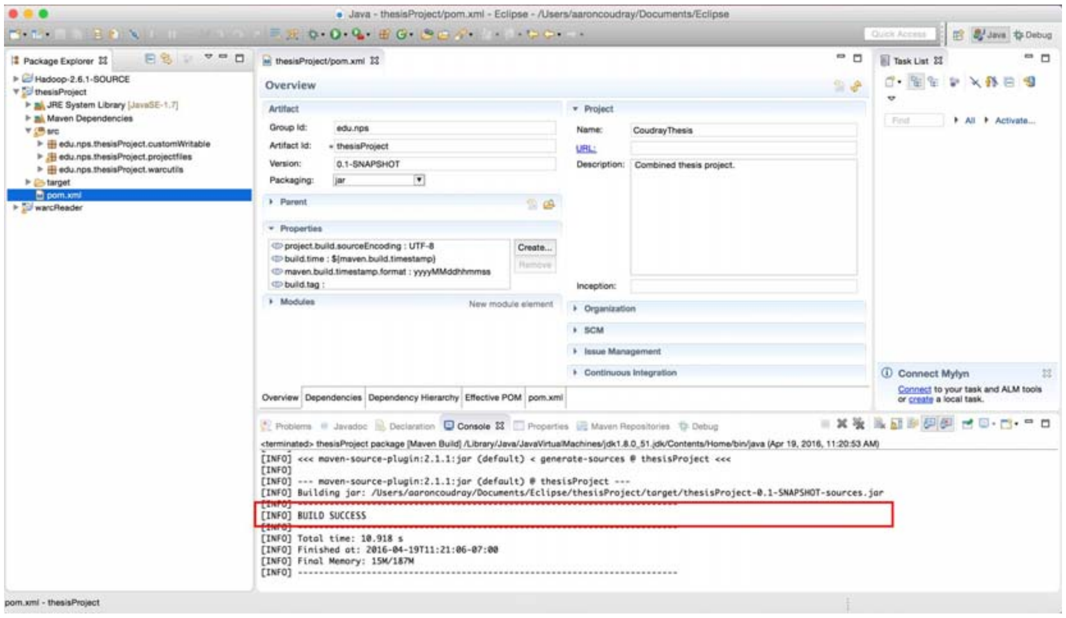
● Append environment to native environment

○ Replace native environment with specified environment

Filter matched 11 of 13 items

Figure 23. Screen Shot of Environment Tab Action for Creating a Custom Maven Build

4. Select Run (see Figure 22). Results of a successful build will look similar to that seen in Figure 24.



Java - thesisProject/pom.xml - Eclipse - /Users/aaroncoudray/Documents/Eclipse

Overview

Artifact

Group Id: edu.nps
Artifact Id: thesisProject
Version: 0.1-SNAPSHOT
Packaging: jar

Properties

project.build.sourceEncoding : UTF-8
build.time : ${maven.build.timestamp}
maven.build.timestamp.format : yyyyMMddhhmmss
build.tag :

Project

Name: CoudrayThesis
URL:
Description: Combined thesis project.
Inception:

```
<terminated> thesisProject package [Maven Build] /Library/Java/JavaVirtualMachines/jdk1.8.0_51.jdk/Contents/Home/bin/java (Apr 19, 2016, 11:20:53 AM)
[INFO] <<< maven-source-plugin:2.1.1:jar (default) < generate-sources @ thesisProject <<<
[INFO]
[INFO] --- maven-source-plugin:2.1.1:jar (default) @ thesisProject ---
[INFO] Building jar: /Users/aaroncoudray/Documents/Eclipse/thesisProject/target/thesisProject-0.1-SNAPSHOT-sources.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 10.918 s
[INFO] Finished at: 2016-04-19T11:21:06-07:00
[INFO] Final Memory: 15M/187M
[INFO] ------------------------------------------------------------------------
```

Figure 24. Screen Shot of Console Illustrating a Successful Build

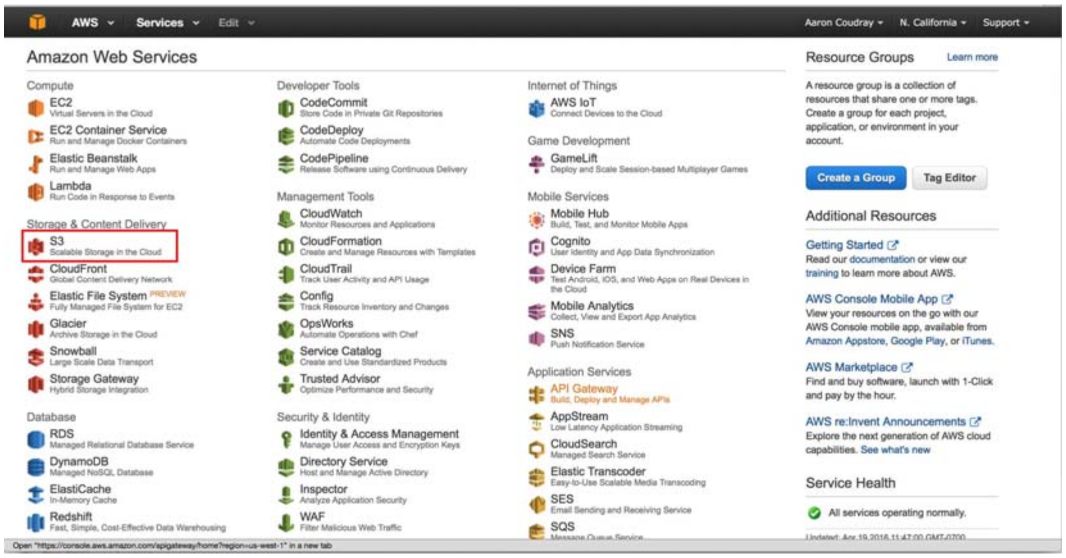
Note: Once this custom Maven build has been created and saved, it will be available for future use. Subsequent project builds will only need to re-perform steps (1) and (4).

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX J. RUNNING A PROJECT ON AWS

In order to run a MapReduce job on AWS, multiple steps must be taken, from creating an S3 bucket, loading a custom JAR file, to creating an EC2 cluster to run that job on. Within this process, there are numerous variable settings that can be selected. The following steps provide guidance on how to run a custom JAR within AWS:

1. Login to AWS.
2. Select: Services -> S3 (see Figure 25).


AWS
Services
Edit
Aaron Coudray
N. California
Support
Amazon Web Services
Compute
EC2 Virtual Servers in the Cloud
EC2 Container Service Run and Manage Docker Containers
Elastic Beanstalk Run and Manage Web Apps
Lambda Run Code in Response to Events
Storage & Content Delivery
S3 Scalable Storage in the Cloud
CloudFront Global Content Delivery Network
Elastic File System PREVIEW Fully Managed File System for EC2
Glacier Archive Storage in the Cloud
Snowball Large Scale Data Transport
Storage Gateway Hybrid Storage Integration
Database
RDS Managed Relational Database Service
DynamoDB Managed NoSQL Database
ElastiCache In-Memory Cache
Redshift Fast, Simple, Cost-Effective Data Warehousing
Developer Tools
CodeCommit Store Code in Private Git Repositories
CodeDeploy Automate Code Deployments
CodePipeline Release Software using Continuous Delivery
Management Tools
CloudWatch Monitor Resources and Applications
CloudFormation Create and Manage Resources with Templates
CloudTrail Track User Activity and API Usage
Config Track Resource Inventory and Changes
OpsWorks Automate Operations with Chef
Service Catalog Create and Use Standardized Products
Trusted Advisor Optimize Performance and Security
Security & Identity
Identity & Access Management Manage User Access and Encryption Keys
Directory Service Host and Manage Active Directory
Inspector Analyze Application Security
WAF Filter Malicious Web Traffic
Internet of Things
AWS IoT Connect Devices to the Cloud
Game Development
GameLift Deploy and Scale Session-based Multiplayer Games
Mobile Services
Mobile Hub Build, Test, and Monitor Mobile Apps
Cognito User Identity and App Data Synchronization
Device Farm Test Android, iOS, and Web Apps on Real Devices in the Cloud
Mobile Analytics Collect, View and Export App Analytics
SNS Push Notification Service
Application Services
API Gateway Build, Deploy and Manage APIs
AppStream Low Latency Application Streaming
CloudSearch Managed Search Service
Elastic Transcoder Easy-to-Use Scalable Media Transcoding
SES Email Sending and Receiving Service
SQS Message Queue Service
Resource Groups
Learn more
Create a Group
Tag Editor
Additional Resources
Getting Started
AWS Console Mobile App
AWS Marketplace
AWS re:Invent Announcements
Service Health

A resource group is a collection of resources that share one or more tags. Create a group for each project, application, or environment in your account.

Read our documentation or view our training to learn more about AWS.

View your resources on the go with our AWS Console mobile app, available from Amazon Appstore, Google Play, or iTunes.

Find and buy software, launch with 1-Click and pay by the hour.

Explore the next generation of AWS cloud capabilities. See what's new

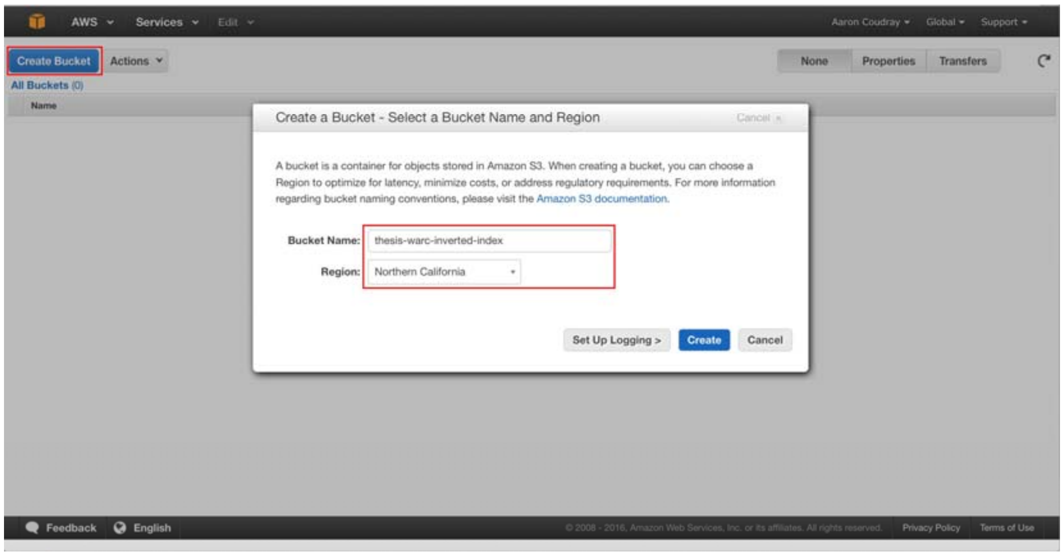
All services operating normally.

Updated: Apr 19 2016 11:47:00 GMT-0700

Open "https://console.aws.amazon.com/apigateway/home?region=us-west-1" in a new tab

Figure 25. Screen Shot of AWS Console with S3 Selected

3. Create bucket: specify Name and Region (see Figure 26).



Create a Bucket - Select a Bucket Name and Region

A bucket is a container for objects stored in Amazon S3. When creating a bucket, you can choose a Region to optimize for latency, minimize costs, or address regulatory requirements. For more information regarding bucket naming conventions, please visit the Amazon S3 documentation.

Bucket Name: thesis-warc-inverted-index

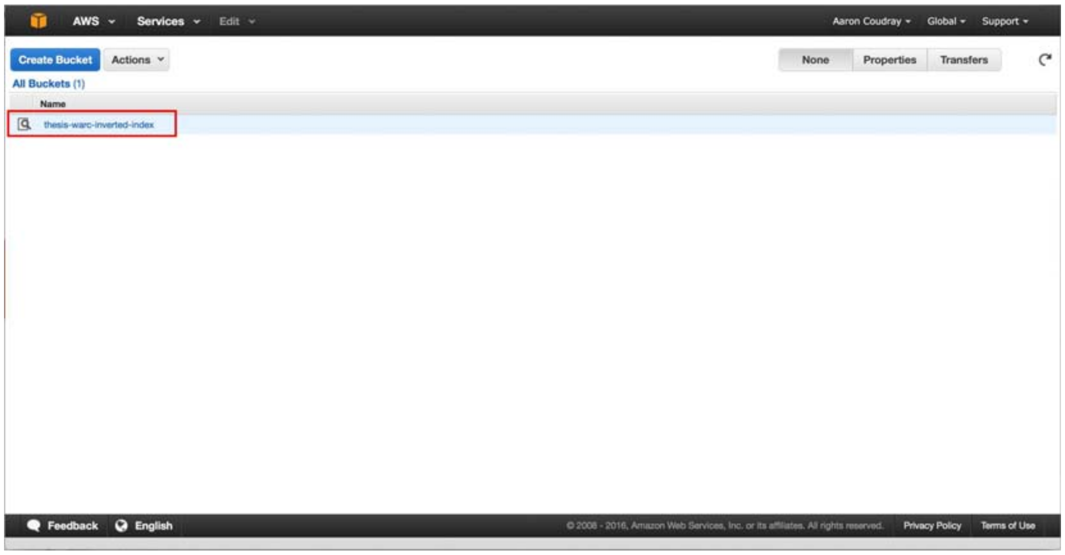
Region: Northern California

Set Up Logging > Create Cancel

© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Figure 26. Screen Shot of S3 Bucket Creation

4. Go to created bucket (see Figure 27).

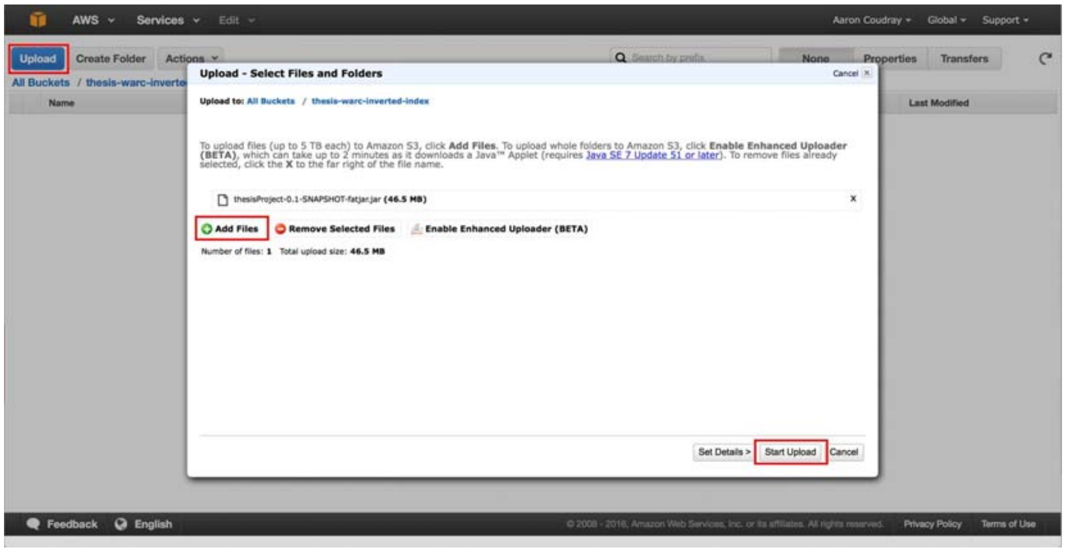


thesis-warc-inverted-index

© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Figure 27. Screen Shot of S3 Bucket Selected

5. Upload JAR to bucket (see Figure 28 and Figure 29).



Upload - Select Files and Folders

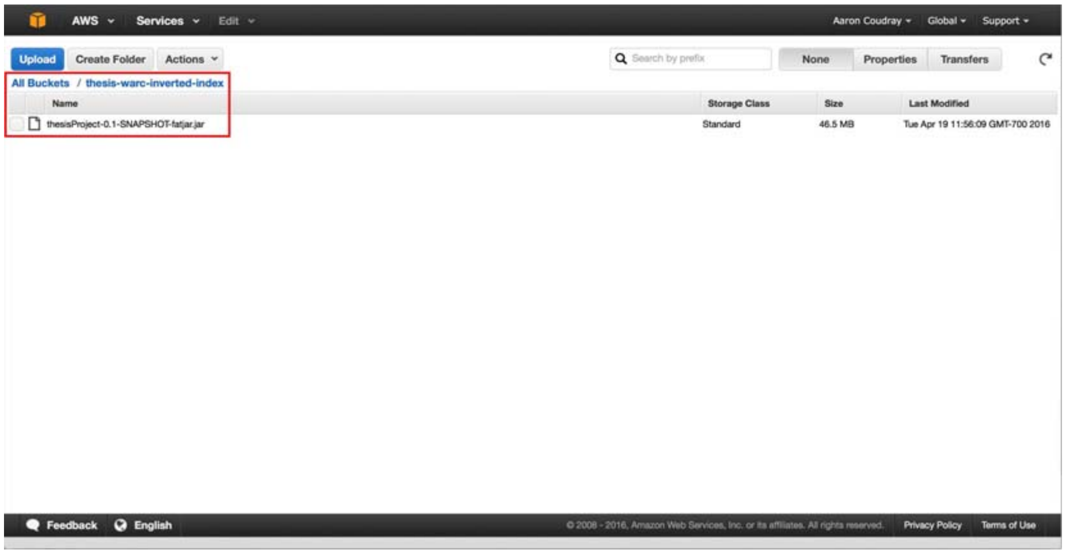
Upload to: All Buckets / thesis-warc-inverted-index

To upload files (up to 5 TB each) to Amazon S3, click **Add Files**. To upload whole folders to Amazon S3, click **Enable Enhanced Uploader (BETA)**, which can take up to 2 minutes as it downloads a Java™ Applet (requires Java SE 7 Update 51 or later). To remove files already selected, click the **X** to the far right of the file name.

thesisProject-0.1-SNAPSHOT-fatjar.jar **(46.5 MB)**

Number of files: **1** Total upload size: **46.5 MB**

Figure 28. Screen Shot of File Upload

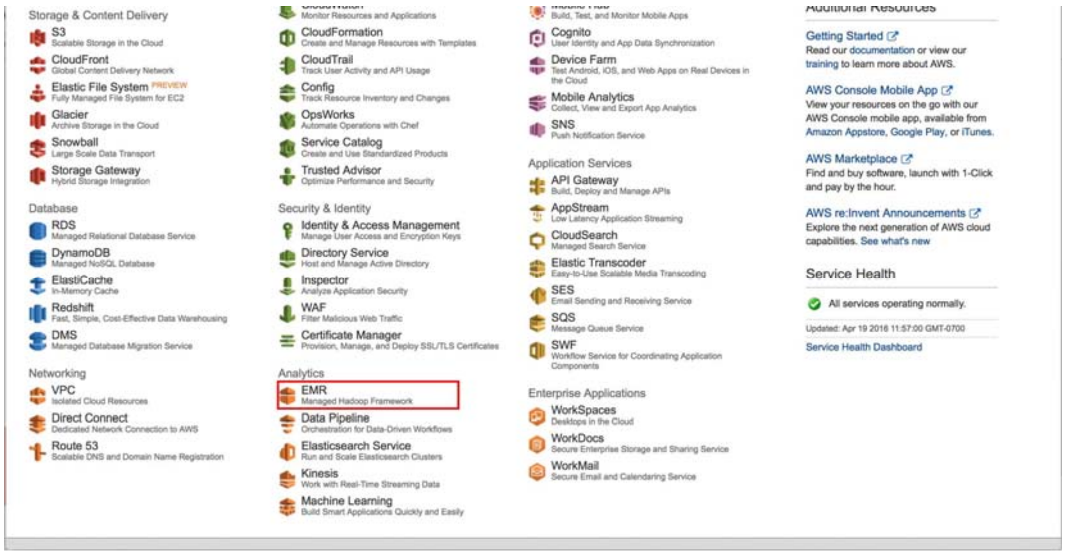


All Buckets / thesis-warc-inverted-index

Name	Storage Class	Size	Last Modified
thesisProject-0.1-SNAPSHOT-fatjar.jar	Standard	46.5 MB	Tue Apr 19 11:56:09 GMT-700 2016

Figure 29. Screen Shot of File in S3 Bucket

6. Select: Services -> EMR (see Figure 30).


Storage & Content Delivery
S3
Scalable Storage in the Cloud
CloudFront
Global Content Delivery Network
Elastic File System PREVIEW
Fully Managed File System for EC2
Glacier
Archive Storage in the Cloud
Snowball
Large Scale Data Transport
Storage Gateway
Hybrid Storage Integration
Database
RDS
Managed Relational Database Service
DynamoDB
Managed NoSQL Database
ElastiCache
In-Memory Cache
Redshift
Fast, Simple, Cost-Effective Data Warehousing
DMS
Managed Database Migration Service
Networking
VPC
Isolated Cloud Resources
Direct Connect
Dedicated Network Connection to AWS
Route 53
Scalable DNS and Domain Name Registration
Monitor Resources and Applications
CloudFormation
Create and Manage Resources with Templates
CloudTrail
Track User Activity and API Usage
Config
Track Resource Inventory and Changes
OpsWorks
Automate Operations with Chef
Service Catalog
Create and Use Standardized Products
Trusted Advisor
Optimize Performance and Security
Security & Identity
Identity & Access Management
Manage User Access and Encryption Keys
Directory Service
Host and Manage Active Directory
Inspector
Analyze Application Security
WAF
Filter Malicious Web Traffic
Certificate Manager
Provision, Manage, and Deploy SSL/TLS Certificates
Analytics
EMR
Managed Hadoop Framework
Data Pipeline
Orchestration for Data-Driven Workflows
Elasticsearch Service
Run and Scale Elasticsearch Clusters
Kinesis
Work with Real-Time Streaming Data
Machine Learning
Build Smart Applications Quickly and Easily
Build, Test, and Monitor Mobile Apps
Cognito
User Identity and App Data Synchronization
Device Farm
Test Android, iOS, and Web Apps on Real Devices in the Cloud
Mobile Analytics
Collect, View and Export App Analytics
SNS
Push Notification Service
Application Services
API Gateway
Build, Deploy and Manage APIs
AppStream
Low Latency Application Streaming
CloudSearch
Managed Search Service
Elastic Transcoder
Easy-to-Use Scalable Media Transcoding
SES
Email Sending and Receiving Service
SQS
Message Queue Service
SWF
Workflow Service for Coordinating Application Components
Enterprise Applications
WorkSpaces
Desktops in the Cloud
WorkDocs
Secure Enterprise Storage and Sharing Service
WorkMail
Secure Email and Calendaring Service
Getting Started
AWS Console Mobile App
AWS Marketplace
AWS re:Invent Announcements
Service Health
Service Health Dashboard

Read our documentation or view our training to learn more about AWS.

View your resources on the go with our AWS Console mobile app, available from Amazon Appstore, Google Play, or iTunes.

Find and buy software, launch with 1-Click and pay by the hour.

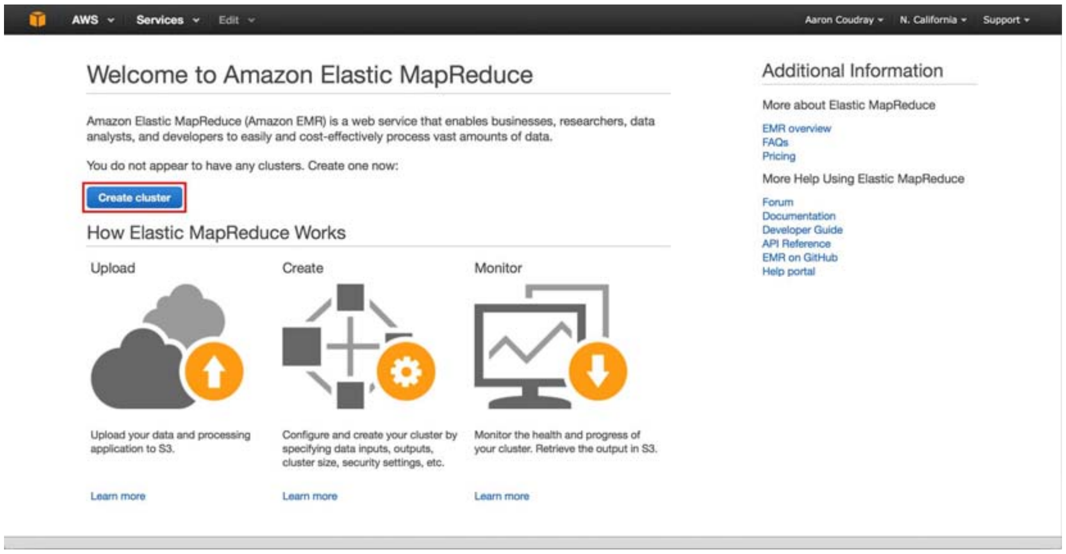
Explore the next generation of AWS cloud capabilities. See what's new

All services operating normally.

Updated: Apr 19 2016 11:57:00 GMT-0700

Figure 30. Screen Shot of AWS Console with EMR Selected

7. Select: Create Cluster (see Figure 31).



Welcome to Amazon Elastic MapReduce

Amazon Elastic MapReduce (Amazon EMR) is a web service that enables businesses, researchers, data analysts, and developers to easily and cost-effectively process vast amounts of data.

You do not appear to have any clusters. Create one now:

How Elastic MapReduce Works

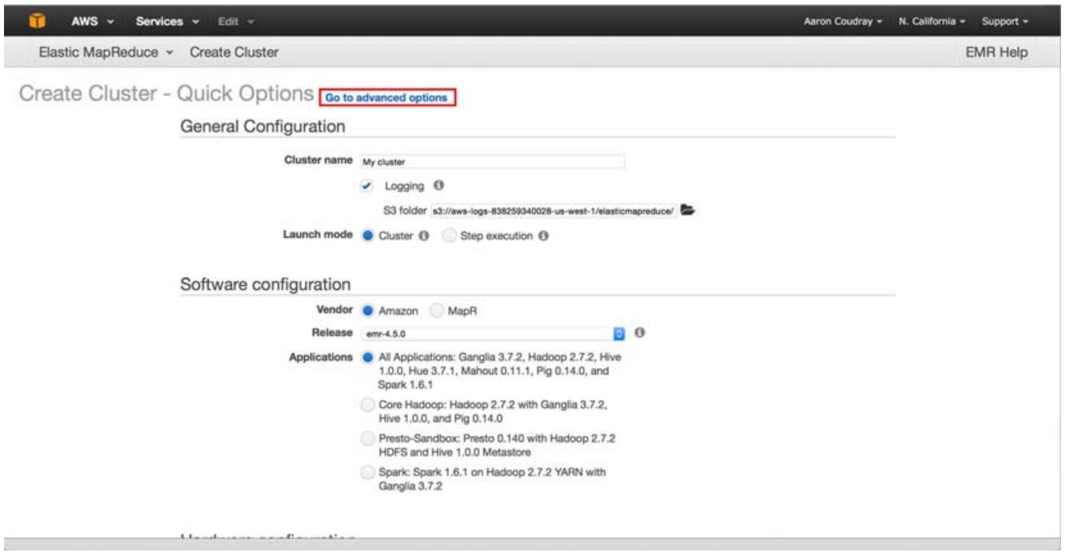
Upload your data and processing application to S3.

Configure and create your cluster by specifying data inputs, outputs, cluster size, security settings, etc.

Monitor the health and progress of your cluster. Retrieve the output in S3.

Figure 31. Screen Shot of Create Cluster Selected

8. Select: Advanced Options (see Figure 32).



Create Cluster - Quick Options
Go to advanced options

General Configuration

Cluster name: My cluster
[✓] Logging
S3 folder: s3://aws-logs-838259340028-us-west-1/elasticmapreduce/
Launch mode: [✓ Cluster] Step execution

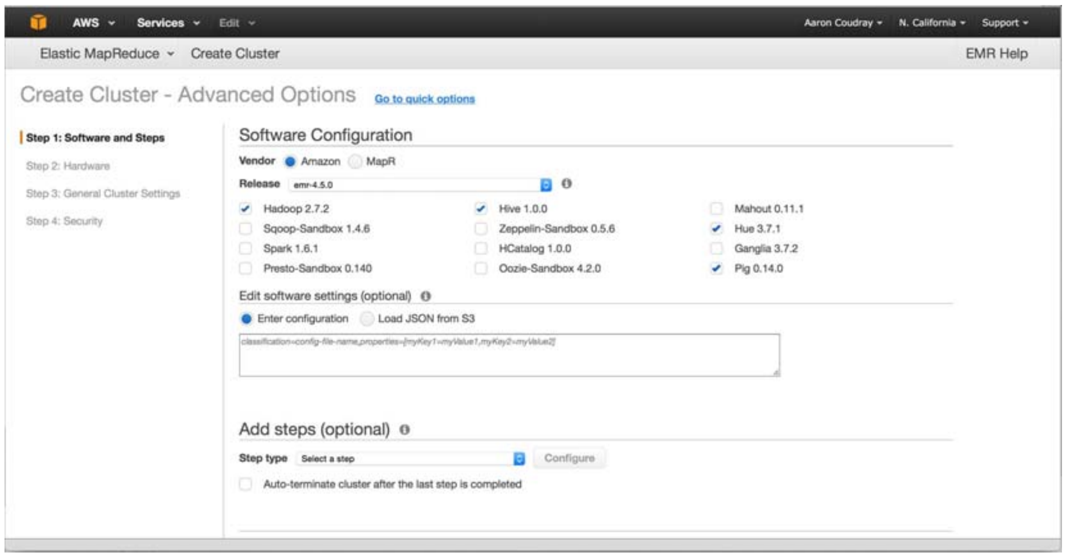
Software configuration

Vendor: [✓ Amazon] MapR
Release: emr-4.5.0
Applications:
[✓ All Applications: Ganglia 3.7.2, Hadoop 2.7.2, Hive 1.0.0, Hue 3.7.1, Mahout 0.11.1, Pig 0.14.0, and Spark 1.6.1]
Core Hadoop: Hadoop 2.7.2 with Ganglia 3.7.2, Hive 1.0.0, and Pig 0.14.0
Presto-Sandbox: Presto 0.140 with Hadoop 2.7.2 HDFS and Hive 1.0.0 Metastore
Spark: Spark 1.6.1 on Hadoop 2.7.2 YARN with Ganglia 3.7.2

Figure 32. Screen Shot of Advanced Options Selected

9. Setup: Software and Steps (see Figure 33). Required:

- Vendor: Amazon
- Release: emr-4.4.0 (or greater)
- Hadoop 2.7.1 (or greater) checked
- Add step: Custom JAR (see step 10).
- Auto-terminate: checked


AWS | Services | Edit | Aaron Coudray | N. California | Support

Elastic MapReduce | Create Cluster | EMR Help

Create Cluster - Advanced Options Go to quick options

Step 1: Software and Steps
Step 2: Hardware
Step 3: General Cluster Settings
Step 4: Security

Software Configuration

Vendor [✓ Amazon] MapR

Release emr-4.5.0

✓ Hadoop 2.7.2	✓ Hive 1.0.0	Mahout 0.11.1
Sqoop-Sandbox 1.4.6	Zeppelin-Sandbox 0.5.6	✓ Hue 3.7.1
Spark 1.6.1	HCatalog 1.0.0	Ganglia 3.7.2
Presto-Sandbox 0.140	Oozie-Sandbox 4.2.0	✓ Pig 0.14.0

Edit software settings (optional)

[✓ Enter configuration] Load JSON from S3

classification=config-file-name,properties=[myKey1=myValue1,myKey2=myValue2]

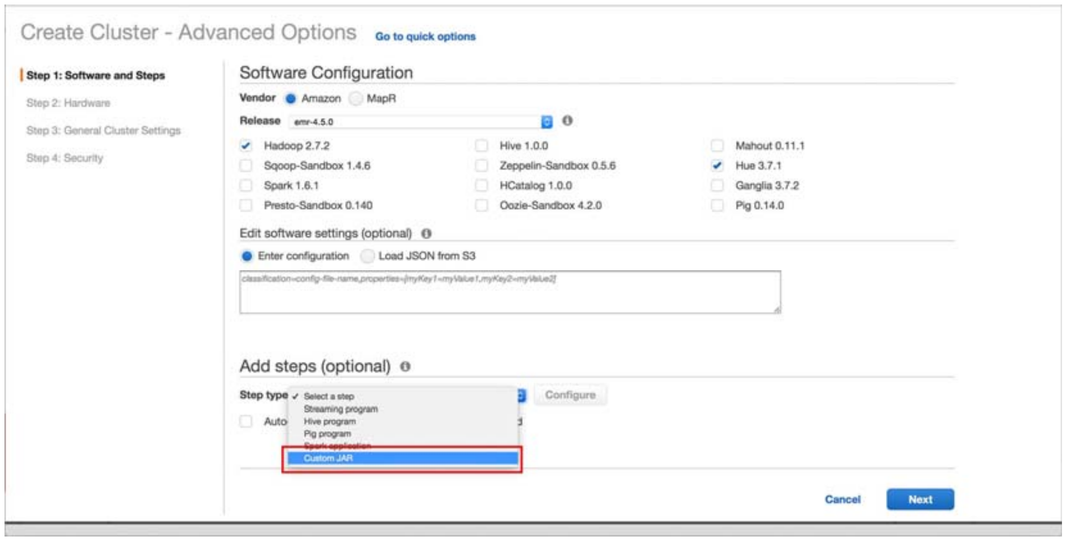
Add steps (optional)

Step type: Select a step | Configure

Auto-terminate cluster after the last step is completed

Figure 33. Screen Shot of Default Software and Steps Page

10. Add Custom JAR Step (see Figure 34).


Create Cluster - Advanced Options Go to quick options

Step 1: Software and Steps
Step 2: Hardware
Step 3: General Cluster Settings
Step 4: Security

Software Configuration

Vendor [✓ Amazon] MapR

Release emr-4.5.0

✓ Hadoop 2.7.2	Hive 1.0.0	Mahout 0.11.1
Sqoop-Sandbox 1.4.6	Zeppelin-Sandbox 0.5.6	✓ Hue 3.7.1
Spark 1.6.1	HCatalog 1.0.0	Ganglia 3.7.2
Presto-Sandbox 0.140	Oozie-Sandbox 4.2.0	Pig 0.14.0

Edit software settings (optional)

[✓ Enter configuration] Load JSON from S3

classification=config-file-name,properties=[myKey1=myValue1,myKey2=myValue2]

Add steps (optional)

Step type: ✓ Select a step / Streaming program / Hive program / Pig program / Spark application / Custom JAR | Configure

Auto

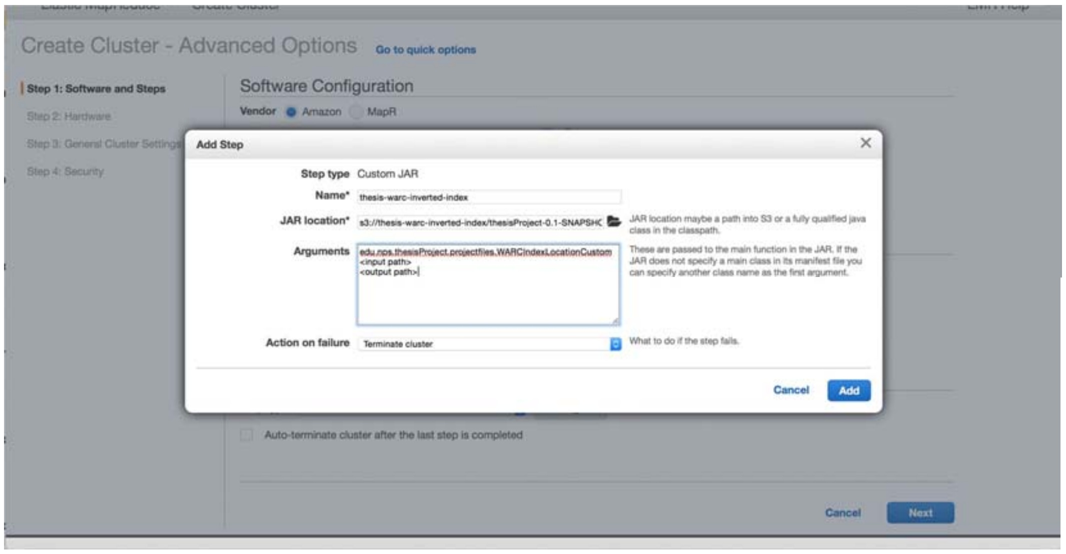
Cancel | Next

Figure 34. Screen Shot of Custom JAR Additional Step Selected

11. Specify JAR steps (see Figure 35).

- Name: (as desired)
- Location: browse to uploaded JAR in S3 bucket (step 5)

- Arguments: <Main Class> <input path> <output path> (as required by JAR)
- Action on failure: Terminate cluster


Create Cluster - Advanced Options Go to quick options

Step 1: Software and Steps
Step 2: Hardware
Step 3: General Cluster Settings
Step 4: Security

Software Configuration

Vendor [✓ Amazon] MapR

Add Step

Step type: Custom JAR
Name*: thesis-warc-inverted-index
JAR location*: s3://thesis-warc-inverted-index/thesisProject-0.1-SNAPSHC — JAR location maybe a path into S3 or a fully qualified java class in the classpath.
Arguments:
```
edu.nps.thesisProject.projectfiles.WARCIndexLocationCustom
<input path>
<output path>
```
These are passed to the main function in the JAR. If the JAR does not specify a main class in its manifest file you can specify another class name as the first argument.
Action on failure: Terminate cluster — What to do if the step fails.

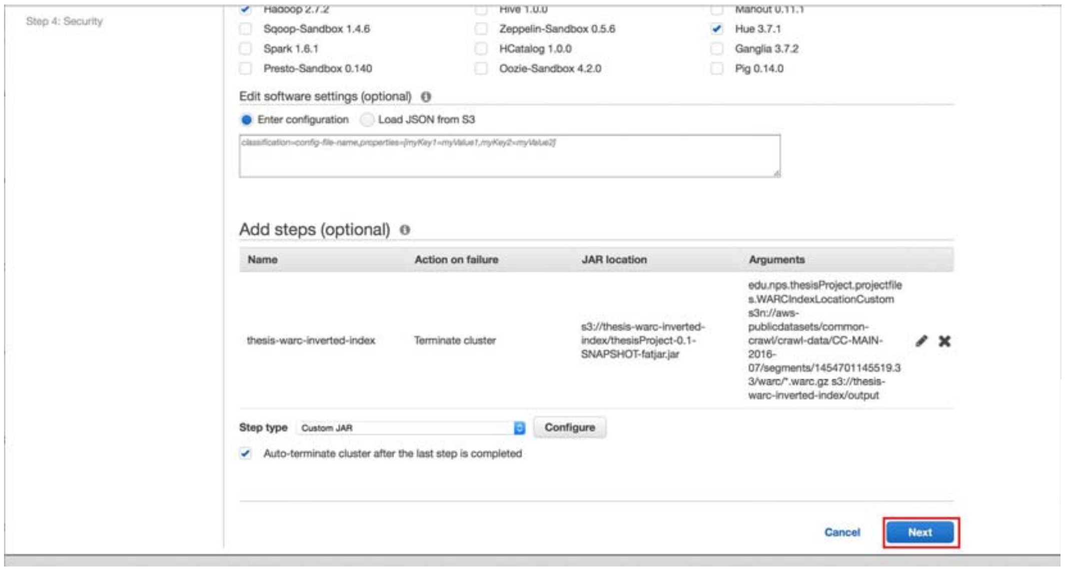
Cancel Add

☐ Auto-terminate cluster after the last step is completed

Cancel Next

Figure 35. Screen Shot of Custom JAR Setup Dialogue

12. Once complete with Steps (9) through (11) select Next (see Figure 36).


Step 4: Security

✓ Hadoop 2.7.2	☐ Hive 1.0.0	☐ Mahout 0.11.1
☐ Sqoop-Sandbox 1.4.6	☐ Zeppelin-Sandbox 0.5.6	✓ Hue 3.7.1
☐ Spark 1.6.1	☐ HCatalog 1.0.0	☐ Ganglia 3.7.2
☐ Presto-Sandbox 0.140	☐ Oozie-Sandbox 4.2.0	☐ Pig 0.14.0

Edit software settings (optional)

[✓ Enter configuration] Load JSON from S3

classification=config-file-name,properties=[myKey1=myValue1,myKey2=myValue2]

Add steps (optional)

Name	Action on failure	JAR location	Arguments
thesis-warc-inverted-index	Terminate cluster	s3://thesis-warc-inverted-index/thesisProject-0.1-SNAPSHOT-fatjar.jar	edu.nps.thesisProject.projectfiles.WARCIndexLocationCustom s3n://aws-publicdatasets/common-crawl/crawl-data/CC-MAIN-2016-07/segments/1454701145519.33/warc/*.warc.gz s3://thesis-warc-inverted-index/output

Step type: Custom JAR Configure

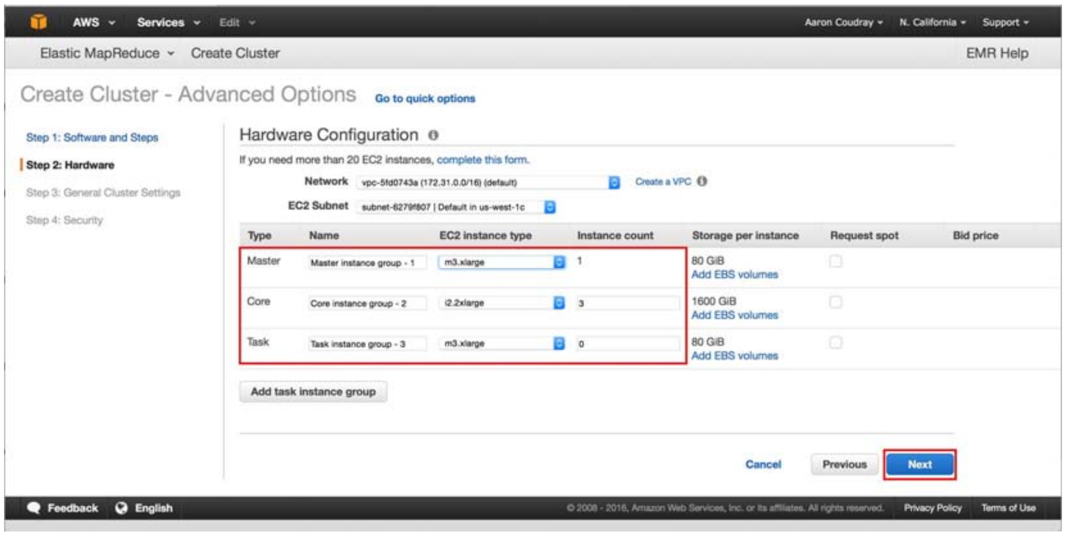
✓ Auto-terminate cluster after the last step is completed

Cancel Next

Figure 36. Screen Shot of Completed Software and Steps Page with Next Selected

13. Setup: Hardware (see Figure 37). Once complete select Next. Required:

- Master Instance
- Core Instance(s)



Create Cluster - Advanced Options

Hardware Configuration

If you need more than 20 EC2 instances, complete this form.

Network: vpc-5fd0743a (172.31.0.0/16) (default)

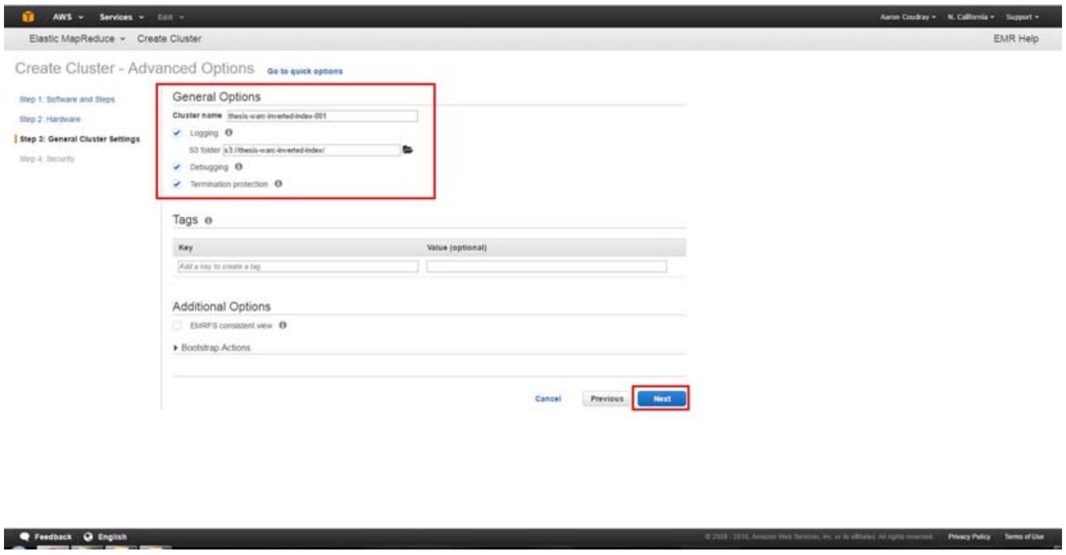
EC2 Subnet: subnet-6279f807 | Default in us-west-1c

Type	Name	EC2 instance type	Instance count	Storage per instance	Request spot	Bid price
Master	Master instance group - 1	m3.xlarge	1	80 GiB Add EBS volumes	☐	
Core	Core instance group - 2	i2.2xlarge	3	1600 GiB Add EBS volumes	☐	
Task	Task instance group - 3	m3.xlarge	0	80 GiB Add EBS volumes	☐	

Figure 37. Screen Shot of Hardware Setup Page with Next Selected

14. Setup: General Cluster Settings (see Figure 38). Once complete select Next. Required:

- Cluster Name: (as desired)
- Logging: (recommended) if selected, choose job S3 bucket
- Debugging: (recommended)
- Termination Protection: (recommended)



General Options

Cluster name: thesis-warc-inverted-index-001

[✓] Logging

S3 folder: s3://thesis-warc-inverted-index/

[✓] Debugging

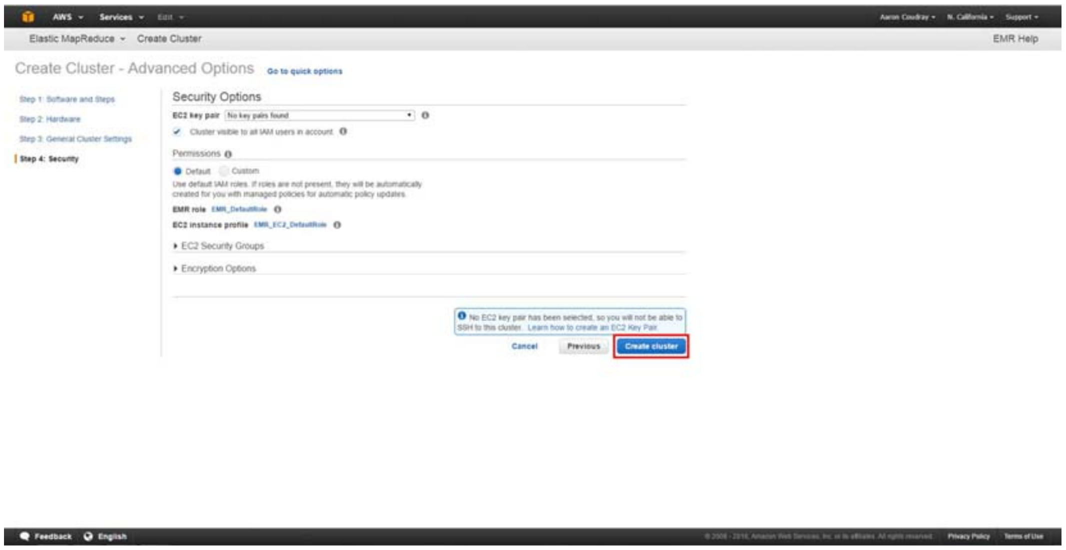
[✓] Termination protection

Key	Value (optional)
Add a key to create a tag	

[ ] EMRFS consistent view

Figure 38. Screen Shot of General Cluster Settings Page with Next Selected

15. Setup: Security Options (see Figure 39). Once complete Create Cluster. No required actions.



Security Options

EC2 key pair: No key pairs found

[✓] Cluster visible to all IAM users in account

Permissions

(● Default) Custom

Use default IAM roles. If roles are not present, they will be automatically created for you with managed policies for automatic policy updates.

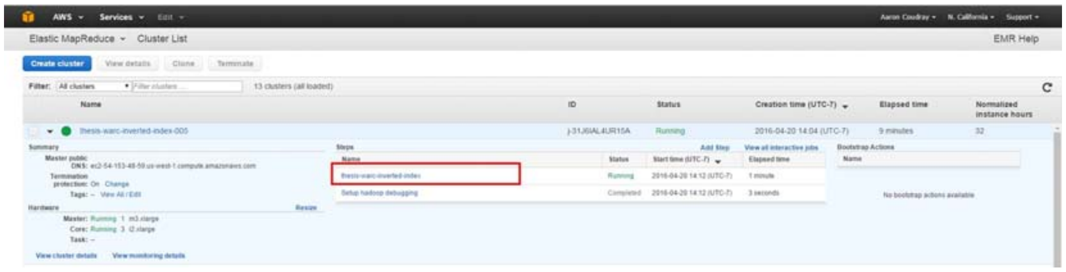
EMR role: EMR_DefaultRole

EC2 instance profile: EMR_EC2_DefaultRole

No EC2 key pair has been selected, so you will not be able to SSH to this cluster. Learn how to create an EC2 Key Pair.

Figure 39. Screen Shot of Default Security Page with Create Cluster Selected

16. A screen shot illustrating a successfully created and running cluster is shown in Figure 40. In order to view log files for job select the JAR step name (see Figure 40).



Filter: All clusters | Filter clusters ... | 13 clusters (all loaded)

Name	ID	Status	Creation time (UTC-7)	Elapsed time	Normalized instance hours
thesis-warc-inverted-index-005	j-31J6IAL4UR15A	Running	2016-04-20 14:04 (UTC-7)	9 minutes	32

Summary
Master public DNS: ec2-54-153-48-59.us-west-1.compute.amazonaws.com
Termination protection: On Change
Tags: -- View All / Edit

Hardware
Master: Running 1 m3.xlarge
Core: Running 3 i2.xlarge
Task: --

View cluster details View monitoring details

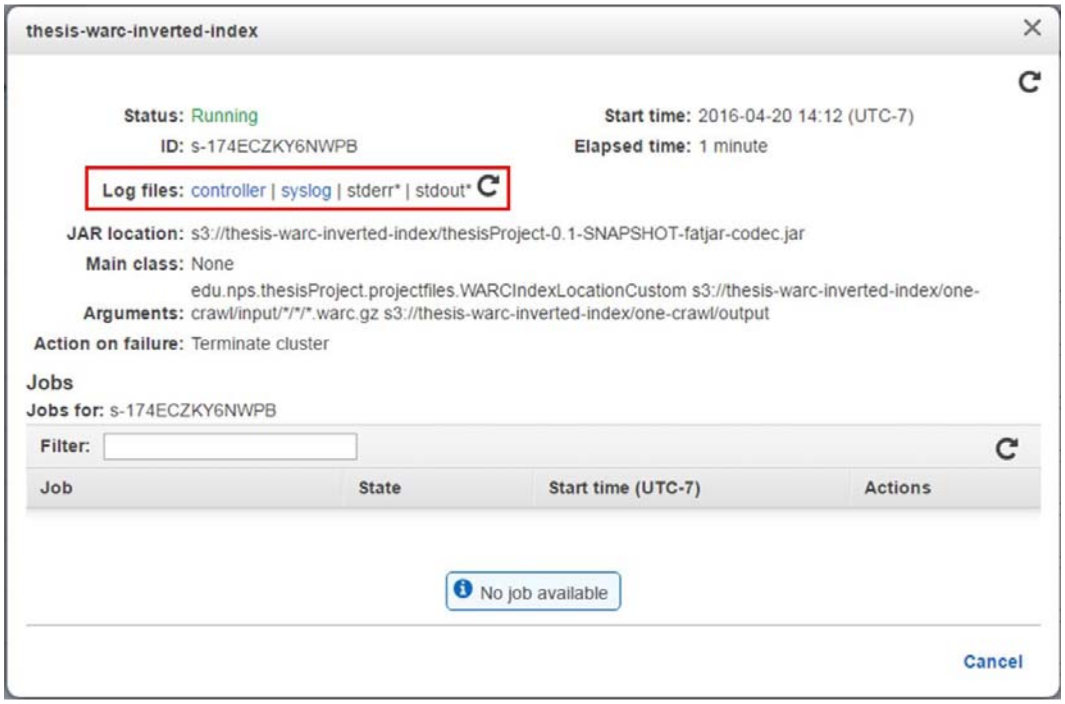
Steps

Name	Status	Start time (UTC-7)	Elapsed time
thesis-warc-inverted-index	Running	2016-04-20 14:12 (UTC-7)	1 minute
Setup hadoop debugging	Completed	2016-04-20 14:12 (UTC-7)	3 seconds

Bootstrap Actions
Name
No bootstrap actions available

Figure 40. Screen Shot of Running Cluster with Custom Step Highlighted

17. A screen shot illustrating JAR step status and log file links is shown in Figure 41.


thesis-warc-inverted-index

Status: Running
ID: s-174ECZKY6NWPB
Log files: controller | syslog | stderr* | stdout*
JAR location: s3://thesis-warc-inverted-index/thesisProject-0.1-SNAPSHOT-fatjar-codec.jar
Main class: None
Arguments: edu.nps.thesisProject.projectfiles.WARCIndexLocationCustom s3://thesis-warc-inverted-index/one-crawl/input/*/*/*.warc.gz s3://thesis-warc-inverted-index/one-crawl/output
Action on failure: Terminate cluster
Start time: 2016-04-20 14:12 (UTC-7)
Elapsed time: 1 minute

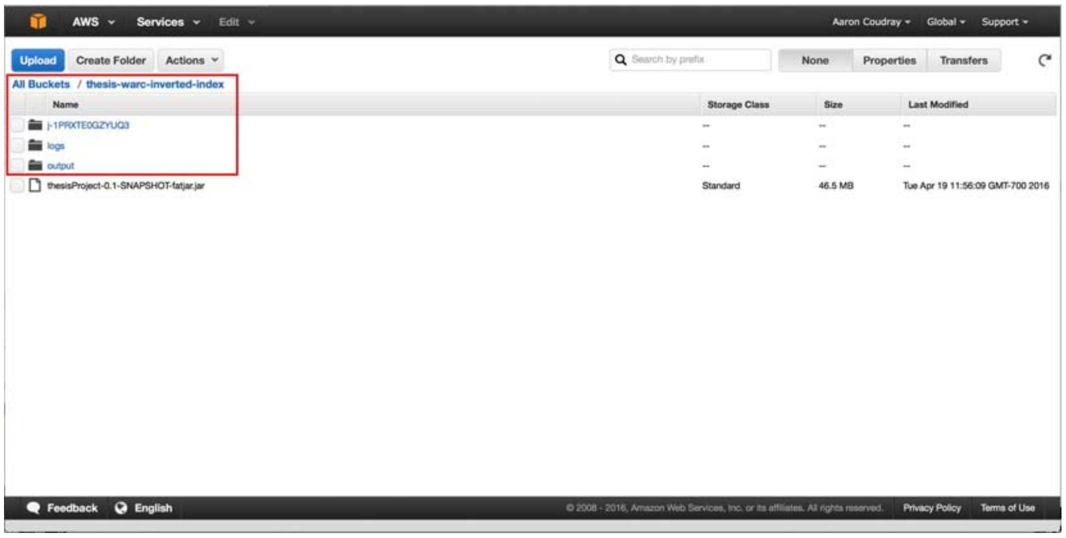
Jobs
Jobs for: s-174ECZKY6NWPB
Filter:

Job	State	Start time (UTC-7)	Actions

No job available

Figure 41. Screen Shot of Custom Step Status with Log Files Highlighted

18. A screen shot illustrating the output of a completed job in its respective S3 bucket is shown in Figure 42.

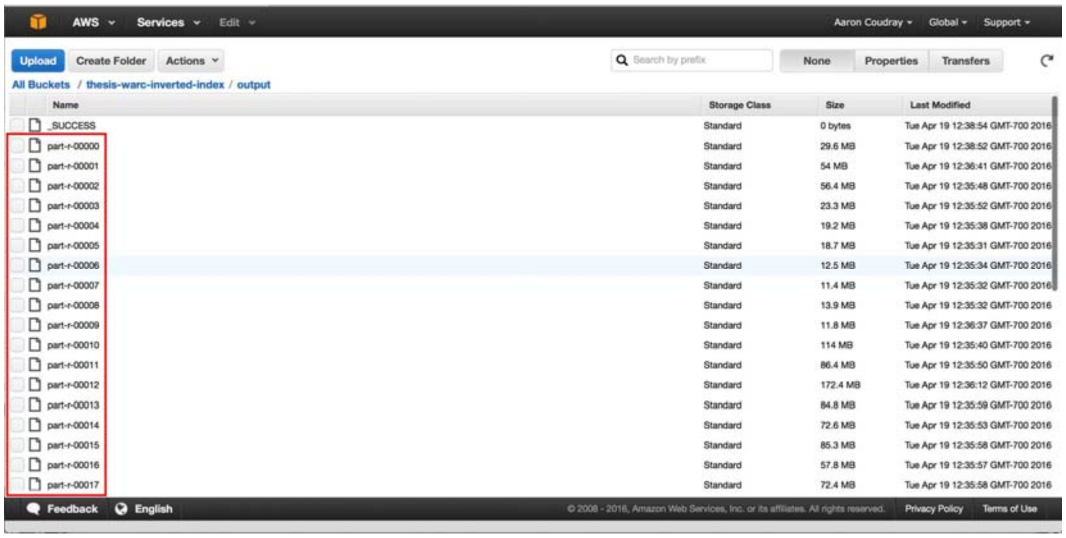


All Buckets / thesis-warc-inverted-index

Name	Storage Class	Size	Last Modified
j-1PRXTE0GZYUQ3	--	--	--
logs	--	--	--
output	--	--	--
thesisProject-0.1-SNAPSHOT-fatjar.jar	Standard	46.5 MB	Tue Apr 19 11:56:09 GMT-700 2016

Figure 42. Screen Shot of S3 Bucket with Output of Completed Job

19. A screen shot illustrating the individual output files is shown in Figure 43.



All Buckets / thesis-warc-inverted-index / output

Name	Storage Class	Size	Last Modified
_SUCCESS	Standard	0 bytes	Tue Apr 19 12:38:54 GMT-700 2016
part-r-00000	Standard	29.6 MB	Tue Apr 19 12:38:52 GMT-700 2016
part-r-00001	Standard	54 MB	Tue Apr 19 12:36:41 GMT-700 2016
part-r-00002	Standard	56.4 MB	Tue Apr 19 12:35:48 GMT-700 2016
part-r-00003	Standard	23.3 MB	Tue Apr 19 12:35:52 GMT-700 2016
part-r-00004	Standard	19.2 MB	Tue Apr 19 12:35:38 GMT-700 2016
part-r-00005	Standard	18.7 MB	Tue Apr 19 12:35:31 GMT-700 2016
part-r-00006	Standard	12.5 MB	Tue Apr 19 12:35:34 GMT-700 2016
part-r-00007	Standard	11.4 MB	Tue Apr 19 12:35:32 GMT-700 2016
part-r-00008	Standard	13.9 MB	Tue Apr 19 12:35:32 GMT-700 2016
part-r-00009	Standard	11.8 MB	Tue Apr 19 12:36:37 GMT-700 2016
part-r-00010	Standard	114 MB	Tue Apr 19 12:35:40 GMT-700 2016
part-r-00011	Standard	86.4 MB	Tue Apr 19 12:35:50 GMT-700 2016
part-r-00012	Standard	172.4 MB	Tue Apr 19 12:36:12 GMT-700 2016
part-r-00013	Standard	84.8 MB	Tue Apr 19 12:35:59 GMT-700 2016
part-r-00014	Standard	72.6 MB	Tue Apr 19 12:35:53 GMT-700 2016
part-r-00015	Standard	85.3 MB	Tue Apr 19 12:35:58 GMT-700 2016
part-r-00016	Standard	57.8 MB	Tue Apr 19 12:35:57 GMT-700 2016
part-r-00017	Standard	72.4 MB	Tue Apr 19 12:35:58 GMT-700 2016

Figure 43. Screen Shot of Reducer Individual Output Files

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX K. SUPPLEMENTAL

Included in this section is a list of all supplemental source code and project files that are available upon request. To obtain a copy of these files, contact the Dudley Knox Library located on the campus of the Naval Postgraduate School in Monterey, California. A copy of this information is included in the project files.

A. ORIGINAL CODE FILES

- basicInvertedIndexCount.java
- basicInvertedLocation.java
- basicWordCount.java
- termLocationCustomWritable.java
- WARCCompressURL.java
- WARCCompressURLMap.java
- WARCExtractContentType.java
- WARCExtractContentTypeMap.java
- WARCIndexLocationCustom.java
- WARCIndexLocationCustomMap.java
- WARCWordCount.java
- WARCWordCountMap.java
- WETIndexCount.java
- WETIndexCountMap.java
- WETIndexLocation.java
- WETIndexLocationMap.java
- WETIndexLocationCustom.java
- WETIndexLocationCustomMap.java
- WETIndexRecordLocation.java

- WETIndexRecordLocationMap.java
- WETWordCount.java
- WETWordCountMap.java
- pom.xml

B. CODE FILES FROM ANOTHER SOURCE

The following files were authored by mathijs.kattenberg@surfsara.nl and obtained from github.com.

- WarcInputFormat.java
- WarcIOConstants.java
- WarcRecordReader.java
- WarcSequenceFileInputFormat.java
- WarcSequenceFileRecordReader.java

LIST OF REFERENCES

[1] X. Wu, X. Zhu, G-Q. Wu, and W. Ding, “Data mining with big data,” *Knowl. and Data Eng, IEEE Trans. On,* vol. 26, pp. 97–107, January 2014.

[2] “Common Crawl,” *Common Crawl.* [Online]. Available: http://commoncrawl.org/. [Accessed: 01-May-2016].

[3] International Organization for Standardization, “Information and documentation - The WARC File Format,” *ISO 28500,* 2009.

[4] Merriam-Webster, “Dictionary and Thesaurus,” 2015. [Online]. Available: http://www.merriam-webster.com/. [Accessed: 01-May-2016].

[5] D. Laney, “3D data management: Controlling data volume, velocity and variety,” *META Group Research Note,* vol. 6, pp. 70, 2001.

[6] V. N. Gudivada, R. Baeza-Yates, and V. V. Raghavan, “Big Data: Promises and Problems,” *Computer,* pp. 20-23, March 2015.

[7] S. Ghemawat, H. Gobioff ,and S. Leung, “The google file system,” in *ACM SIGOPS Operating Systems Review,* 2003, pp. 29-43.

[8] T. White, *Hadoop : The Definitive Guide.* 4th ed. Sebastopol, CA: O'Reilly, 2015.

[9] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters, OSDI’04: Sixth symposium on operating system design and implementation, San Francisco, CA, December, 2004.

[10] “Welcome to Apache™ Hadoop®!,” 2014. [Online]. Available: http://hadoop.apache.org/. [Accessed: 01-May-2016].

[11] “Intro to Hadoop and MapReduce | Udacity,” *Udacity.com*, 2011 [Online]. Available: https://www.udacity.com/course/intro-to-hadoop-and-mapreduce--ud617. [Accessed: 01-May-2016].

[12] “Apache Nutch™,” 2014. [Online]. Available: https://nutch.apache.org/. [Accessed: 01-May-2016].

[13] “Amazon Web Services (AWS) – Cloud Computing Services,” *Amazon Web Services*, 2016 [Online]. Available: https://aws.amazon.com/. [Accessed: 01-May-2016].

[14] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce.* San Rafael, CA: Morgan & Claypool, 2010.

[15] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval.* Cambridge university press Cambridge, 2008.

[16] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: bringing order to the web." 1999.

[17] R. Muesal, P. Petrovski, and C. Bizer, "The WebDataCommons microdata, RDFa and microformat dataset series," in *13th International Semantic Web Conference,* Riva del Garda, Italy, 2014, pp. 277-292.

[18] C. Bizer, K. Eckert, R. Meusel, H. Mühleisen, M. Schuhmacher, and J. Völker, "Deployment of rdfa, microdata, and microformats on the web–a quantitative analysis," in *The Semantic Web–ISWC,* Springer Berlin Heidelberg, 2013, pp. 17-32.

[19] "Amazon S3 Tools: Command Line S3 Client and S3 Backup for Windows, Linux: s3cmd, s3express," 2009. [Online]. Available: http://s3tools.org/s3cmd. [Accessed: 01-May-2016].

[20] "Cyberduck | Libre FTP, SFTP, WebDAV, S3, Backblaze B2 & OpenStack Swift browser for Mac and Windows," *Cyberduck*. [Online]. Available: https://cyberduck.io/. [Accessed: 01-May-2016].

[21] "Eclipse–The Eclipse Foundation open source community website," 2016. [Online]. Available: https://eclipse.org/. [Accessed: 01-May-2016].

[22] B. Porter, J. van Zyl, and O. Lamy, "Maven–Welcome to Apache Maven," 2002. [Online]. Available: https://maven.apache.org/. [Accessed: 01-May-2016].

[23] "Sandbox–Hortonworks," *Hortonworks*, 2011. [Online]. Available: https://hortonworks.com/products/sandbox/. [Accessed: 01-May-2016].

[24] "Cloudera Enterprise Downloads," *Cloudera*. [Online]. Available: http://www.cloudera.com/downloads.html. [Accessed: 01-May-2016].

[25] "Oracle VM VirtualBox," *Virtualbox.org*. [Online]. Available: https://www.virtualbox.org/. [Accessed: 01-May-2016].

[26] "JAR File Overview," 1993. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jarGuide.html. [Accessed: 01-May-2016].

[27] E. Redmond and K. H. Marbaise, "Maven–POM Reference," 2002. [Online]. Available: https://maven.apache.org/pom.html. [Accessed: 01-May-2016].

[28] "iipc/webarchive-commons," *GitHub*, 2016. [Online]. Available: https://github.com/iipc/webarchive-commons. [Accessed: 01-May-2016].

[29] “JWAT–Java Web Archive Toolkit–SBForge Confluence,” *Sbforge.org*. [Online]. Available: https://sbforge.org/display/JWAT/JWAT. [Accessed: 01-May-2016].

[30] “Lesson: Regular Expressions (The Java™ Tutorials > Essential Classes),” 1995. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/regex/. [Accessed: 01-May-2016].

[31] Internet Engineering Task Force, “Hypertext Transfer Protocol - HTTP/1.1,” *RFC 2616*, 1999.

[32] J. Hedley, “jsoup Java HTML Parser, with best of DOM, CSS, and jquery,” 2009. [Online]. Available: https://jsoup.org/. [Accessed: 01-May-2016].

[33] “Amazon Elastic MapReduce (EMR)–Amazon Web Services,” *Amazon Elastic MapReduce*, 2016. [Online]. Available: https://aws.amazon.com/elasticmapreducep/. [Accessed: 01-May-2016].

[34] “Java heap space–Plumbr,” 2016. [Online]. Available: https://plumbr.eu/outofmemoryerror/java-heap-space. [Accessed: 01-May-2016].

[35] “Tuning Java Virtual Machines (JVMs),” 2016. [Online]. Available: https://docs.oracle.com/cd/E13222_01/wls/docs81/perform/JVMTuning.html. [Accessed: 01-May-2016].

[36] “Elastic Cloud Compute (EC2) Cloud Server & Hosting–AWS,” *Elastic Cloud Compute*, 2016. [Online]. Available: https://aws.amazon.com/ec2/. [Accessed: 01-May-2016].

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California